MASTER EN SCIENCES INFORMATIQUES

Application et évaluation d'une méthodologie de programmation logique

Mignon, Bertrand

Award date:
1988

Awarding institution:
Universite de Namur

Application et évaluation d'une méthodologie de programmation logique

Bertrand Mignon

Mémoire réalisé en vue
de l'obtention du titre
de Licencié et Maître
en Informatique

Promoteur : Baudoin Le Charlier

Namur Septembre 1988

abstract: Le but de ce mémoire est d'appliquer et d'évaluer une méthodologie de programmation logique [Deville 87]. La méthodologie est d'abord présentée. Ensuite une application est réalisée à l'aide de celle-ci. L'application sert à transformer des programmes prolog pour les rendre plus performants (déclaration de mode). La méthodologie est finalement évaluée et des extensions et modifications sont proposées.

abstract: The aim of this thesis is to use and evaluate a methodology for logic programs construction. First the methodology is presented. Then an application to transform prolog programs to optimize them (mode declaration) is realised with the methodology. Finally, the methodology is evaluated and modifications and extensions are proposed.

REMERCIEMENTS

Je tiens à remercier Messieurs Baudouin Le Charlier et Yves Deville pour leurs commentaires, leurs conseils et leur patience lors de la réalisation de ce mémoire.

Je remercie également le personnel de la K.U.Leuven et plus particulièrement Messieurs Maurice Bruynooghe et Dany de Schreye pour leur accueil et leur aide lors de mon stage.

Je désire également associer à ces remerciements les professeurs et assistants de l'Institut d'Informatique ainsi que mes camarades de classe pour cinq années d'études et de joie.

Ces remerciements ne sauraient être complets sans parler de mes parents et de Sylviane, ma femme, qui m'ont soutenu et aidé pendant mes études.

Table des matières

Introduction

En 1972, Kowalski et Colmerauer eurent l'idée fondamentale que **la logique peut être utilisée comme langage de programmation** [Kowalski 74]. Peu de temps après le langage Prolog (pour **Prog**rammation **Log**ique) était né.

Depuis lors, la popularité de la programmation logique n'a cessé de croître et de nombreuses recherches, qui ne se limitent pas à Prolog, y ont été consacrées dans des domaines tels que: les fondations théoriques de la programmation logique, la logique et les bases de données, les applications (systèmes experts), les implémentations, les méthodologies, ...

Dans ce mémoire, nous allons nous intéresser à une méthodologie de construction de programmes logiques [Deville 87]. Cette méthodologie couvre les différentes phases de construction d'un programme, de la spécification jusqu'à l'obtention d'un programme prolog correct et efficace. La principale caractéristique de cette méthodologie est de découper la construction d'une procédure en deux phases. Construction d'un algorithme logique pendant laquelle l'aspect déclaratif de prolog est pris en compte. Dérivation d'une procédure prolog et optimisation de celle-ci, phase pendant laquelle la sémantique procédurale de prolog est prise en compte.

Cette méthodologie va être utilisée pour réaliser une application en Prolog. Cette application a pour sujet la transformation de programmes Prolog afin de les rendre plus performants. La réalisation de cette application a été effectuée lors d'un stage à la K.U.Leuven et s'intègre à un projet de recherche sur la compilation d'informations de contrôle, [Bruynooghe 86].

La construction de l'application a un double but : tester si la méthodologie est appropriée pour construire un programme d'une certaine taille et évaluer la méthodologie grâce à l'expérience acquise.

Le mémoire est structuré comme suit. La méthodologie de construction de programmes logiques est présentée au chapitre 1. Ensuite le chapitre 2 décrit le cadre général de

l'application réalisée : compilation d'informations de contrôle et principes de transformation de programmes Prolog pour générer des déclarations de mode. Le chapitre 3 est consacré au développement de l'application selon la méthodologie proposée. Le chapitre 4 contient une évaluation de l'application réalisée tandis que le chapitre 5 est consacré à évaluer la méthodologie. Le texte complet de l'application ainsi que le mode d'emploi se trouvent en annexe.

CHAPITRE 1

Présentation de la méthodologie

La méthodologie qui va être présentée se base sur la thèse de doctorat d'Yves Deville. Nous nous limiterons volontairement dans la présentation à certains aspects de la thèse. Pour une présentation complète le lecteur intéressé peut se référer au texte original [Deville 87].

Cette méthodologie permet de construire à partir d'une spécification informelle, un programme logique correct et efficace. Le langage cible choisi est PROLOG, le programme logique obtenu est donc un programme PROLOG.

L'idée de base de la méthodologie est de diviser l'effort de construction du programme en trois phases.

La première est la phase de spécification. La spécification d'un programme comprend la description d'une relation en langage naturel et l'ensemble des conditions d'application du programme.

La seconde phase a pour but de construire un algorithme logique correct à partir de la spécification. Cette phase s'occupe uniquement de l'aspect déclaratif du langage et est fortement indépendante du langage cible.

La dernière phase s'occupe elle de l'aspect procédural de PROLOG. Tout ce qui fait que programmer en PROLOG est différent de programmer en logique est examiné. L'algorithme logique est transformé en une procédure logique aussi efficace que possible. Ces tranformations se font de manière à conserver la correction de l'algorithme de départ.

Ce chapitre sera découpé selon les trois phases de construction du programme.

1.1 SPECIFICATION.

Après un bref rappel de quelques notions, la forme générale et standard d'une spécification est présentée.

1.1.1 Rappel.

Une **substitution** θ est un ensemble fini de la forme $\{V1/t1,...,Vk/tk\}$ où chaque Vi est une variable, ti un terme distinct de vi, et les variables vi sont toutes distinctes.

Si E est une expression, **Eθ l'instance de E selon θ**, est l'expression obtenue en remplaçant simultanément chaque occurrence d'une variable vi de E par le terme ti ($1 \leq i \leq k$). [Lloyd 87].

Une **procédure logique P d'arité n, noté P/n**, est une séquence de clauses qui ont toutes le même foncteur principal P d'arité n dans leur tête.

Un **terme de base** est un terme qui ne contient pas de variable.

1.1.2 Forme générale d'une spécification.

procédure p(T1,T2,...,Tn).

Soit
T1 un type1,
.. ...
Tn un typen.

pre: autres préconditions.

Cette procédure détermine

une relation p existe entre T1,T2,...,Tn.

directionnalité:
In(.....) Out(.....) <Min,Max>
.....

préconditions de l'environnement: .....

effets secondaires: .....

1.1.2.1 Types et préconditions.

type$_i$ est le nom d'un ensemble non vide (ex:entiers positifs, listes,..) dont tous les éléments sont des termes de base.

type$_i^*$ est l'ensemble des termes qui ont une instance qui appartient à type$_i$.

De manière formelle, $A \in type_i^*$ ssi $\exists\ \theta : A\theta \in type_i$.

exemple:

[b,o,n,j,o,u,r] ∈ list,
[H| T] ∈ list*.

(avec H et T deux variables).

On remarque que quelque soit type$_i$ si X est une variable alors X ∈ type$_i$*.

La déclaration de type joue un double rôle :

- elle précise la forme d'un paramètre actuel, Ai, après l'exécution de la procédure: si θ est une substitution réponse calculée par la procédure, alors Aiθ ∈ type$_i$*. La déclaration de type est dans ce sens une postcondition.
- elle restreint la forme des paramètres actuels, Ai, avant l'exécution de la procédure: ∀ i, Ai ∈ typei*, ou encore que <A1,A2,...,An> ∈ (type1xtype2x...xtype$_n$)*. En ce sens la déclaration de type est une précondition. Si cette précondition n'est pas remplie alors l'exécution de la procédure a un effet indéterminé.

Les autres préconditions sont un ensemble de relations entre les paramètres formels (ex:Ti>Tj). Leur signification est semblable à la signification des types :

- elles donnent des informations sur la forme des Ai après l'exécution de la procédure.
- elles restreignent la forme des Ai avant l'exécution de la procédure : au moins une instance de base doit respecter les préconditions avant l'exécution de la procédure sinon la procédure a un effet indéterminé.

Dans la suite par précondition on se réfère aussi bien aux préconditions sur les types qu' aux autres préconditions.

Même si PROLOG est un langage de programmation non typé, ajouter une restriction sur le type que peut prendre un paramètre d'une procédure est intéressant à plusieurs points de vue :

- cela permet d'exprimer la relation qui existe entre les différents paramètres de manière plus simple car seuls les cas qui respectent les préconditions doivent être envisagés.
- la construction d'un programme est aussi simplifiée. Si une relation n'a de sens que pour des paramètres d'un type donné et que l'on désire construire un algorithme pour des paramètres de n'importe quel type, des tests doivent être ajoutés à l'intérieur de la procédure pour qu'elle ait un sens et soit réalisable. Il semble plus simple et plus efficace d'exprimer ces restrictions comme des préconditions et d'ajouter des tests à l'extérieur de la procédure uniquement quand cela est nécessaire.
- cela permet de connaître le type d'un paramètre après l'exécution d'une procédure.
- un certain nombre de primitives de Prolog sont typées, la déclaration de type est indispensable pour pouvoir les spécifier correctement.

1.1.2.2 Relation.

Une relation est un ensemble de tuples de base <a1,....,an>.

La propriété qui caractérise cet ensemble de couples est exprimée de manière simple et précise en langage naturel.

Quand la relation ne peut être exprimée directement de manière simple, alors des concepts intermédiaires doivent être définis afin d'avoir une description simple et claire de la relation.

Quand les paramètres de la procédure ne sont pas des termes de base, la procédure détermine s'il existe une instance de base des paramètres telle que <A1θ,....,Anθ> ∈ relation.

1.1.2.3 Directionnalité.

La **directionnalité** décrit les usages possibles de la procédure et le nombre de substitutions réponses associées.

Un **usage possible** de la procédure décrit la forme des paramètres actuels, Ai, avant l'exécution de la procédure (partie In) et de Aiθ (θ est une substitution réponse) après l'exécution de la procédure (partie Out).

Le **nombre de substitutions réponses** (<Min,Max>) indique une borne inférieure et une borne supérieure au nombre de substitutions réponses qui seront effectivement calculées pour un usage possible de la procédure.

Trois valeurs possibles sont retenues pour exprimer la forme d'un paramètre : variable **var**, terme de base **gr** (pour ground), ni terme de base ni variable **ngv** pour (neither ground nor var).

Les bornes inférieures et supérieures peuvent prendre les valeurs suivantes: un entier positif, infini (∞) ou un nombre positif fini mais inconnu (n).

Une directionnalité est notée par la forme que doivent avoir les paramètres en entrée, In(m1,.....,mn) et la forme correspondante en sortie Out(M1,...,Mn), le nombre de réponses par <min,max>.
avec - mi,Mi =/= {},
- mi,Mi inclus {gr,var,ngv},
- min ∈ N U {∞},
- max ∈ N U {∞,n}.

Pour des facilités d'écriture et de lecture {f} sera noté f.

On définit aussi les ensembles suivants:
novar = { gr,ngv},
nogr = { var,ngv},
nongv = { var,gr},
any = { var,gr,ngv}

La directionnalité de la procédure p est un ensemble de directionnalités.

exemple:

In(gr,var,any) Out(gr,gr,ngv) <1,∞>
In(nogr,gr,ngv) Out(ngv,gr,ngv) <0,n>

Les paramètres A1,...An respectent la directionnalité si chaque Ai a une forme ∈ mi et AiΘ a une forme ∈ Mi (Θ une substitution réponse) et min ≤ le nombre de substitutions réponses ≤ max.

Ecrire une procédure multidirectionnelle n'est pas toujours possible, il est donc intéressant d'indiquer les seuls usages possibles de la procédure. La partie In est donc une précondition à l'utilisation de la procédure. Si la précondition n'est pas respectée l'effet de la procédure est indéterminé.

Connaître la forme des paramètres en sortie et le nombre de substitutions réponses associées étant donné leur forme en entrée est indispensable pour pouvoir écrire une procédure PROLOG correcte qui utilise d'autres procédures dont la forme des paramètres en entrée n'est pas quelconque, et pour établir la terminaison de cette procédure.

1.1.2.4 Préconditions de l'environnement et effets de bord.

Par préconditions de l'environnement on entend les aspects non logiques dont peut dépendre la bonne exécution de la procédure, ex: fichiers, entrée/sortie, etc.

Les effets de bords (side effects) d'une procédure sont souvent liés à la sémantique procédurale de PROLOG, les décrire n'est pas simple surtout quand le nombre de substitutions réponses est plus grand que un.

Comme nombre de procédures n'ont pas d'effet de bord l'omission de cette partie signifie qu'il n'y a pas d'effet de bord.

1.1.2.5 Exemple.

procédure functor(T,F,N).

Soit T un terme,
F un atom,
N un entier positif.

Cette procédure détermine si

T est un terme d'arité N dont le foncteur principal est F.

directionnalité.
In(novar,any,any) Out(novar,gr,gr) <0,1>
In(var,gr,gr) Out(ngv,gr,gr) <1,1>

1.1.3 Remarques.

Avoir une spécification correcte et précise pour une procédure est très important.

Si la spécification provient de la phase d'analyse des besoins elle est la seule garantie que le problème que l'on va résoudre est bien celui qu'on doit résoudre.

Lors de la construction d'un programme il est fréquent de définir des sous-problèmes plus simples à résoudre et de construire le programme à l'aide de ces sous-problèmes (méthode top-down). Une spécification correcte des sous-problèmes est indispensable pour pouvoir montrer que le programme est correct.

1.2 ALGORITHME LOGIQUE.

Après avoir spécifié une procédure, deux grandes étapes sont nécessaires pour obtenir un programme PROLOG correct: construire un algorithme logique correct et le transformer en un programme prolog. La méthode proposée permet d'obtenir des algorithmes logiques corrects par construction et est basée sur le principe d'induction structurelle. La possibilité de généraliser le problème pour pouvoir le résoudre plus facilement ou plus efficacement est aussi envisagée. Finalement un système de transformation d'algorithmes logiques est présenté.

1.2.1 Construction d'un algorithme logique.

La méthode présentée ici se limite aux programmes qui ont besoin d'utiliser la récursivité (n'importe quelle sorte de boucle) pour être résolus. Les autres programmes sont souvent faciles à construire de fait même qu'ils n'ont pas besoin d'utiliser la récursivité. Pour des raisons d'efficacité il est parfois utile de quand même développer une solution récursive (ex tri par permutation).

1.2.1.1 Construction top-down.

La construction top-down d'un algorithme consiste à résoudre cet algorithme en termes de sous-problèmes qui ont été spécifiés. Ce processus s'arrête quand on obtient un problème primitif.

Un problème primitif dépend du langage cible. En PROLOG on a par exemple des primitives d'entrée sortie, de comparaison, des opérations arithmétiques, etc.... Toutes ces primitives sont spécifiées dans le manuel d'utilisation du langage de manière plus ou moins précise et peuvent varier d'une implémentation à l'autre. En programmation logique la primitive la plus importante est sans aucun doute l'égalité "=".

1.2.1.2 Méthode de construction par induction structurelle.

La méthode présentée permet d'obtenir des algorithmes corrects par construction. La correction des algorithmes est basée sur l'équivalence entre la relation décrite dans la spécification et les conséquences logiques d'un ensemble d'algorithmes, mais uniquement pour les termes qui respectent les préconditions des spécifications. La définition précise de la correction d'un programme est basée sur la notion de modèle et plus spécialement de modèle d'Herbrand. Avec un peu d'intuition il est cependant possible de s'en passer dans beaucoup de cas.

A. Concepts.

Soit E un ensemble.
$<$ une relation binaire dans E.

Une séquence $x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots$ d'éléments de E est une **séquence décroissante** ssi $x_1 > x_2 > \ldots > x_{i-1} > x_i > x_{i+1} > \ldots$

La **relation** $<$ **est bien fondée** dans E, ou $(E,<)$ est un ensemble bien fondé ssi il n'y a pas de séquence décroissante infinie d'éléments de E.

Un élément e de E est un **élément minimal** de E ssi il n'y a pas d'élément e' de E tel que $e' < e$.

exemple :

-l'ensemble des entiers positifs est bien fondé pour la relation "est plus petit que".
-l'ensemble des listes est bien fondé pour les relations:
L1 < L2 ssi L1 est la queue de L2
L1 < L2 ssi L1 a moins d'éléments que L2.
-l'ensemble des arbres est bien fondé pour la relation T1 < T2 ssi T1 est un sous-arbre de T2.

Principe d'induction:

Soit $(E,<)$ un ensemble bien fondé,
W(x) une propriété des éléments de E,

A partir de $\forall x \in E$: si $\forall y < x : W(y)$
conclure $\forall x \in E$: W(x).

Application:

Soit W(x) une propriété des éléments de E.

Le principe d'induction s'applique d'habitude de la manière suivante :

-on définit une relation bien fondée < dans E.
-on prouve que W(x∘) est vrai pour tout élément minimal de E.
-pour x non minimal on prouve que W(x) est vrai en supposant que W(y) est vrai pour certains y<x.
-on conclut que W(x) est vrai pour tous les éléments de E.

B. Forme générale d'un algorithme.

Les algorithmes construits avec la méthode ont la forme suivante:

V X1,X2,...,Xn (p(X1,X2,...,Xn) <=> F).
avec n≥0 , et F une formule.
F à la forme (∃ Y1,...,Yk) C1 & F1
V .....
V Cm & Fm
avec Yi =/= Xj **V** i,j et Y1,..Yk est l'ensemble des variables libres des Ci & Fi.

Etant donné la forme de nos algorithmes nous écrirons dans la suite la même formule de la manière suivante:

p(X1,X2,....,Xn) <=> C1 & F1
V .....
V Cm & Fm
en n'oubliant pas que des quantificateurs sont implicitement présents.

exemple:

V L ordered(L) <=> ∃ X, ∃ Y, ∃ T
L = []
/ V L = [X]
V L = [X,Y¦T] & X<Y & ordered([Y¦T])
s'écrit
ordered(L) <=>
L = []
V L = [X]
V L = [X,Y¦T] & ordered([Y¦T]).

C. Processus de construction.

Soit une procédure p(x1,...,Xm), les différentes étapes de construction d'un algorithme logique, LA(P), sont les suivantes:

1. choisir un paramètre Xj que l'on appelle le paramètre d'induction.

2. définir une relation bien fondée pour le type de Xj.

3. construire les Ci à partir des différentes formes structurelles de Xj. Un des Ci doit couvrir le cas minimum et quelle que soit la forme structurelle de Xj, il doit exister un Ci qui couvre cette forme structurelle.

4. Soit p la relation décrite dans la spécification de P. Les Fi sont construits de manière à ce que (Ci & Fi) soit une condition nécessaire et suffisante pour qu'une instance de **base des paramètres qui respectent les préconditions** soit ∈ p quand la forme structurelle de Xj est décrite par Ci. Fi peut être construit en réduisant le problème en sous-problèmes plus simples et/ou par un usage récursif de p(T1,...,Tn) si Tj<Xj selon la relation bien fondée. Si on utilise P pour résoudre un des sous-problèmes (procédure mutuellement récursive) une preuve séparée de la terminaison du programme doit être réalisée. Il faut prouver que l'on se rapproche bien du cas minimal selon la relation bien fondée.

Exemple 1.

Illustrons ce processus par un exemple dont voici la spécification:

procédure member(E,L).

Soit E un terme,
L une liste,

Cette procédure détermine si

E est un élément de la liste L.

1. Choix du paramètre d'induction, Xj.

Ici le choix se limite à L car on doit définir une relation bien fondée sur le type de Xj ce qui n'est pas évident à faire pour un terme quelconque.

2. Choix de la relation bien fondée.

Le choix de la relation bien fondée dépend des différentes formes structurelles que l'on veut mettre en évidence en (3). Une relation simple pour les listes est : L1 < L2 ssi L1 est un suffixe propre de L2.

3.Choix des Ci.

Un choix simple est:

C1:L=[] (cas minimal)
C2:L=[H¦T] (couvre tous les autres cas)

On voit que notre relation bien fondée est bien choisie car dans le cas non minimal on a T < L.

4.Constuction des Fi.

Analysons d'abord le cas minimal: L = []. Quand L est vide, E ne peut manifestement pas être un élément de la liste, on a F1: false.

Quand la liste n'est pas vide notre choix pour C2 nous indique d'essayer de résoudre le problème en fonction du premier élément et du reste de la liste.

Deux cas peuvent se présenter:

-H, le premier élément de L, est E. Dans ce cas E est élément de L.
-le premier élément de L n'est pas E et une condition nécessaire et suffisante pour que E soit élément de la liste, L, est qu'il soit élément du reste de la liste, T. Etant donné le choix de la relation bien fondée, T<L on peut donc appliquer le principe d'induction et utiliser la procédure member pour voir si E fait partie de T. Il faut aussi vérifier que les préconditions de member sont respectées ce qui ne pose ici aucun probléme car T est une liste.
On a F2: E=H
V E =/= H & member(E,T).

N.B. : par convention on notera E=/=H pour ¬ E=H.

On obtient finalement l'algorithme logique suivant:

```
member( E, L) <=>
          L=[] & false
    V     L=[ H¦ T] & ( H=E
                  V     H =/= E & member( E, T) )
```

Ce petit exemple permet déjà de remarquer que les différents choix, paramètre d'induction, relation bien fondée, Ci, ne sont pas indépendants, que ces choix déterminent l'algorithme obtenu (construction des Fi), que le cas minimal doit se résoudre sans le principe d'induction (c'est évident puisqu'il n'existe pas d'élément plus petit selon la relation bien fondée), et que pour les cas non minimaux on n'est pas obligé d'utiliser le principe d'induction (ex: E=H).

Exemple 2.

Le paramètre d'induction est choisi dans l'intention de pouvoir appliquer le principe d'induction lors de la construction de l'algorithme. Il doit être tel que sa structure reflète bien le problème à résoudre et que la relation puisse être facilement exprimée en fonction de cette structure.

Comme dans member/2, il n'y avait pas de choix, prenons la procédure append/3 qui permet de voir l'influence du paramètre d'induction sur l'algorithme obtenu.

procédure append(L1, L2, L_app).

Soit
L1,L2,L_app des listes.

Cette procédure détermine si

L_app est la concaténation de la liste L1 et L2.

Ici L1, L2, L_app sont candidats. Un premier choix peut être de choisir L_app, ce choix nous amène à penser la relation comme "quelles sont les deux listes qui concaténées l'une à l'autre donnent la liste L_app". Si L_app est vide le problème est facile. Si L_app n'est pas vide il reste à trouver si le premier élément de L_app vient de L1 ou de L2.

En prenant "est un suffixe propre" comme relation bien fondée, on obtient assez facilement l'algorithme suivant:

```
append( L1, L2, L_app) <=>
     L_app=[] & L1=[] & L2=[]
  V  L_app=[ H¦ T_app] & ( L1=[] & L2=L_app
                      V    L1=[ H1¦ T1] & H=H1 &
                           append(T1,L2,T_app) )
```

On voit que si on choisit L_app, on doit de toute manière examiner la forme structurelle de L1. Ne serait-il pas plus facile de choisir L1 ? Dans ce cas on perçoit plutôt la relation comme "quelle liste est la concaténation de L1 et de n'importe quelle autre liste". En prenant toujours la relation bien fondée "est un suffixe propre" on obtient l'algorithme bien connu:

```
append( L1, L2, L_app) <=>
     L1=[] & L_app=L2
   V L1=[ H¦ T] & L_app=[ H¦ T_app] & append(T,L2,T_app).
```

Ce second algorithme permet comme on pouvait s'y attendre de construire un algorithme logique pour append plus facilement. On remarque également que ces deux algorithmes sont logiquement équivalents.

Rien ne permet de dire à priori que certains paramètres sont plus mauvais que d'autres, la seule règle à respecter dans tous les cas (sauf bien sûr si ce n'est vraiment pas possible) est de choisir un paramètre qui est toujours de base dans la partie In(...) de la spécification de la directionnalité. Ce choix non seulement facilitera la transformation de l'algorithme logique en une procédure logique, mais il semble également que les algorithmes ainsi construits permettent d'obtenir des procédures logiques plus efficaces.

1.2.3 Généralisation

Pour certains problèmes, la méthode que nous venons de présenter ne suffit pas, soit qu'il n'est pas possible de construire un algorithme correct ou que l'algorithme construit ne permet pas d'obtenir une procédure logique efficace dans la seconde phase de construction du programme.

La solution proposée ici est de généraliser le problème initial pour pouvoir le résoudre.

Deux types de généralisations sont proposées. La première généralise la structure d'un paramètre (généralisation structurelle), nous nous limiterons à la généralisation d'un terme par une liste de termes (tupling generalisation). La seconde (généralisation d'un état de calcul) caractérise un état général de calcul en fonction de ce qui a déjà été fait et de ce qui reste à faire.

1.2.3.1 Généralisation structurelle.

A. Principe.

Soit la procédure p(X,Y) une procédure pour laquelle on veut écrire un algorithme.

procédure p(X,Y).

Soit X un type_X,
Y une liste.

Cette procédure détermine si

<X,Y> ∈ p

La spécification d'une "tupling generalisation" peut être schématisée comme suit :

procédure p_gen(List_X,Y)

Soit
List_X une liste d'objets de type type_x, [X1,..Xn]
Y une liste.

Cette procédure détermine si

Y est la concaténation de Y1,...Yn avec <Xi,Yi> ∈ p (1≤i≤n)

Quand une relation bien fondée est définie sur type_x il est possible de l'étendre à une liste d'objets de type type_x.

Principe:
Soit L1,L2 des listes de type_x,
L1 <L2 ssi L1 peut être obtenu à partir de L2
-en enlevant au moins un élément de L2
-en ajoutant n'importe quel nombre d'éléments de type type_x qui soient plus petits qu'un des éléments enlevés (selon la relation bien fondée sur type_x).

B. Exemple.

Je vais construire une version généralisée q_sort_g/2 de la procédure q_sort/2.

B.1 Spécification.

procédure q_sort(L,L1)

Soit L,L1 des listes d'entiers positifs.

Cette procédure détermine si

L1 est la liste L triée par ordre croissant

Voici la spécification de la procédure généralisée :

procédure q_sort_g(Lg,L1).

Soit
Lg une liste de listes d'entiers positifs [L1,...,Ln],
L1 une liste d'entiers positifs.

Cette procédure détermine si

L1 est la concaténation de Y1,...,Yn avec Yi la liste Li triée par ordre croissant, ($1 \leq i \leq n$).

B.2 Construction.

Je vais construire la procédure par induction sur Lg.

La relation bien fondée est basée sur la relation "contient moins d'éléments" pour les éléments de Lg. Détaillons là plus précisément :

Soit :
- LL1, LL2 deux listes de listes,
- $<$ une relation bien fondée sur les listes (e.g. contient moins d'élément.

LL1 $<^*$ LL2, ($<^*$ une relation bien fondée) ssi LL1 est la liste LL2 à laquelle :
- on a retiré au moins un élément, (soit A1,...,Am, m>0)
- on a ajouté des éléments, (soit B1,...,Bn, $n \geq 0$) tel que chacun de ces éléments est plus petit qu'un des éléments retirés ($\forall$ i, $1 \leq i \leq n$ $\exists$ j, $1 \leq j \leq m$: Bi $<$ Aj).

Si Lg=[] alors L1=[],

Si Lg=[H|T] examinons les cas possibles:

-si H est la liste vide alors **V** T1 tel que q_sort_g(T,T1), L1 est la concaténation de la liste vide et de T1, L1=T1.
-si H n'est pas la liste vide, je vais me servir de q_sort_g/2 pour trier H.
  -Si H contient au moins deux éléments, il est possible de séparer H en trois listes (éventuellement vide) H1, H2, H3 tel que H1<H, H2<H, H3<H selon la relation bien fondée "est un suffixe propre". q_sort_g/2 peut être utilisé car [H1,H2,H3|T] < [H|T].
  -sinon H=[X] et [X] est la liste H triée. Il est alors facile d'obtenir L1 : **V** T1 tel que q_sort_g(T,T1), L1=[X|T1].

En utilisant la procédure partition/3 dont je donnerai ensuite la spécification on obtient les algorithmes suivants:

```
q_sort(L,L1) <=> q_sort_g([L],L1)

q_sort_g(Lg,L1) <=>
     Lg=[] & L1=[]
 V   Lg=[H|T] & ( H=[X] & q_sor_g(T,T1) & L1=[X|T1]
          V      H=[X1,X2|Th] & partition([X2|Th],X1,L,B) &
                 q_sort_g([L,[X1],B|T],L1) ).
```

procédure partition(L,E,Little,Big).

Soit L,Litle,Big des listes d'entiers positifs,
E un entier positif.

Cette procédure détermine si

Little est la liste des éléments de L < E, Big est la liste des éléments de L ≥ E.

L'algorithme obtenu ressemble à l'algorithme de tri "quick sort", il ne nécessite néanmoins qu'un seul appel récursif et il n'est pas nécessaire de concaténer des listes. Il faut cependant gérer des listes de listes.

C. Discussion.

Dans certain cas, il est nécessaire de généraliser la procédure pour pouvoir la construire. Dans d'autre cas, comme dans l'exemple, la généralisation n'est pas nécessaire mais elle permet de remplacer plusieurs appels récursifs par un seul, ce qui peut permettre d'optimiser la procédure logique dérivée.

1.2.3.2 Généralisation d'un état du calcul.

La seconde manière de généraliser un problème est la caractérisation d'un état de calcul.

A. Principe.

Supposons que l'on désire construire un algorithme logique pour p(X,Y).

procédure p(X,Y).

Soit X un type_x,
Y un type_y.

Cette procédure détermine si

<X,Y> ∈ p

Si X est le paramètre d'induction et qu'une relation bien fondée est définie sur type_x, il est possible de représenter n'importe quelle instance de base de X, selon la relation bien fondée comme suit:

X: e1 e2 ... e_i e_{i+1} ... e_n
avec e_{i+1} ... e_n < e_i e_{i+1} ... e_n

La nature de ei dépend de la relation bien fondée. Si X est une liste, cette décomposition est triviale pour la relation "est un suffixe propre", quand X est un entier, on peut prendre 1 2 .. i i+1 ...

Un algorithme contruit avec X comme paramètre d'induction va considérer successivement les différentes formes de X selon la description faite plus haut. Il doit donc être possible de caractériser un état général de calcul en termes de ce qui a déjà été fait et de ce qui reste à faire. De manière schématique:

X: e_{i+1} ... e_n | e_i e_{i+1} ... e_n
Pref_X Suf_X
Int_Y
déjà fait à faire
où Pref_X est e_{i+1} ... e_n
Suf_X est e_i e_{i+1} ... e_n
Int_Y est un résultat partiel correct <Pref_X,Int_Y> ∈ p

Voici maintenant la forme générale de la spécification d'une généralisation de calcul.

procédure p_gen_comp(Suf_X, Y, Int_Y, Pref_X).

Soit Pref_X,Suf_X des type_x,
Y,int_y des type_y.

pre: il existe un terme X de type type_x tel que
-X est Pref_X <> Suf_X
-<Pre_X,Int_y> ∈ p
(avec <> l'opérateur de composition approprié dans type_x).

Cette procédure détermine si

<X,Y> ∈ p.

p(X,Y) est un cas particulier de p_gen_comp s'il existe Øx et Øy tels que
- X est Øx <> X (neutre à gauche)
- <Øx,Øy> ∈ p

Dans ce cas on a:

p(X,Y) <=> p_gen_comp(X,Y,Øx,Øy).

Quand on construit un algorithme pour p_gen_comp, on va essayer de réduire Suf_X à quelque chose de plus petit selon la relation bien fondée et utiliser le principe d'induction. On s'arrête lorsque Suf_X est minimal, à ce moment on a que X est Pre_X et <Pre_X,Int_Y> ∈ p, si Y est Int_Y alors <X,Y> ∈ p et le problème est résolu.

Pre_X sert à étendre Int_Y quand on réduit Suf_X. Souvent on a seulement besoin d'informations à propos de Pre_X et dans certains cas on peut complètement se passer de Pre_X. Cela nous amène à distinguer deux types de généralisations, ascendante et descendante.

Une **généralisation est descendante** quand aucune information à propos de Pre_X n'est nécessaire pour construire p_gen_comp.

Une **généralisation est ascendante** quand on a besoin d'informations à propos de Pre_X pour construire p_gen_comp.

B. Généralisation descendante.

B.1 Exemple.

procédure reverse(L,Lr).

Soit L,Lr deux listes

Cette procédure détermine si

Lr est la liste L inversée.

Si on choisit L comme paramètre d'induction et "est un suffixe propre" comme relation bien fondée, un état de calcul est représenté par Pre_L, Int_Lr et Suf_L.
Schématiquement on a:

L: e1,e2, ...,ei ¦ ei+1, ... Lr: ...,ei+1 ¦ ei,...,e2,e1
Pre_L Suf_L Int_Lr

Si Suf_L est vide on a que Lr est Int_Lr. Si Suf_L n'est pas vide on aimerait bien passer à un état de calcul ou Suf_L est plus petit selon notre relation bien fondée. Schématiquement on veut:

L: e1,e2, ...,ei+1 ¦ ei+2, ... Lr: ...,ei+2 ¦ei+1,...,e2,e1
Pre_L Suf_L Int_Lr

On remarque que seul le premier élément de Suf_L est nécessaire pour changer d'état, Pre_L est donc inutile pour généraliser reverse(L,Lr).

La spécification de reverse_gen peut également être simplifiée.

procédure reverse_gen(L,Int_Lr,Lr).

Soit L,Int_Lr,Lr des listes

Cette procédure détermine si

Lr est la liste L inversée, concaténée à Int_Lr.

En prenant L comme paramètre d'induction et "est un suffixe propre" comme relation bien fondée on obtient facilement, vu ce qui a été dit plus haut, l'algorithme suivant:

reverse_gen(L,Int_Lr,Lr) <=>
    L=[] & Lr=Int_Lr
  V L=[H¦ T] & Int1=[H¦ Int_Lr] & reverse_gen(T,Int1,Lr)

Reverse est bien un cas particulier de reverse_gen car <[],[]> ∈ p et L=[]<>L avec <> l'opérateur de concaténation de liste:

reverse(L,Lr) <=> reverse_gen(L,[],Lr).

B.2 Discussion.

Quand on peut écrire un algorithme pour une généralisation descendante il est possible d'écrire un algorithme pour le problème initial puisqu'aucune information n'est nécessaire sur ce qui a déjà été fait. Quel est alors l'intérêt de cette méthode? Dans la partie suivante nous verrons que les programmes logiques dérivés d'une généralisation sont souvent 'tail recursive' alors que les programmes dérivés du problème initial ne le sont pas.

C. Généralisation ascendante.

C.1 Exemple.

concept.

La fonction de Fibonacci est définie comme suit:

f(0) = 1,
f(1) = 1,
f(x) = f(x-1)+f(x-2).

spécification.

procédure fibonacci(F,N).

Soit F,N deux entiers positifs.

Cette procédure détermine si

F=f(N) avec f la fonction de Fibonacci.

Construction.

Un état de calcul peut être représenté schématiquement comme suit:

```
 0 1 2 ... i ¦ i+1 i+2 ... N
  Pre_N        Suf_N
  Int_F
```

Spécification de la procédure généralisée.

procédure fibonacci_gen(F,N,I,Fi,Fi1).

Soit F,N,Fi,Fi1 des entiers positifs

pre: - 1≤I≤N,
- Fi = f(I).
- Fi1=f(I-1)

Cette procédure détermine si

F= f(N) (f la fonction de Fibonacci).

L'information à propos de Pre_N est Fi1, Suf_N est représenté par (N,I), I≤N correspond à la condition "X est Pre_X <> Suf_X" et Int_N est f(I) ou Fi. On doit imposer I≥1 sinon F(I-1) n'est pas défini.

Construction:

Regardons comment étendre l'état de calcul:

1. Suf_N est vide correspond à la condition I=N dans ce cas F=Fi.

2. I<N. On étend le résultat à I+1:
I1 is I+1, F(I+1) is Fi + Fi1

toutes les conditions sont remplies pour pouvoir employer fibonacci_gen/5 récursivement: 1≤I1≤N, Fnew=f(I1) et Fi=f(I1-1).

Algorithme.

```
fibonacci_gen(F,N,I,Fi,Fi1) <=>
     N=I & F=Fi
  V  I<N & Fnew is Fi + Fi1 & I1 is I + 1 &
     fibonacci_gen(F,N,I1,Fnew,Fi).
```

Quand N≥1 et que Pre_N est vide (I=1) fibonnaci/2 est un cas particulier de fibonnaci_gen/5. En traitant explicitement le cas où N est 0 on obtient

```
fibonacci(F,N) <=>
     N=0 & F=1
  V  N =/= 0 & fibonacci_gen(F,N,1,1,1).
```

C.2 Discussion.

L'information à propos de Pre_x est nécessaire pour réduire Suf_x. La généralisation développée correspond à un calcul de bas en haut (bottom-up computation).

1.2.4 Transformations.

Après avoir construit un algorithme logique il faut encore le transformer pour obtenir un programme logique. Dans certains cas il est impossible d'obtenir un programme efficace à partir de l'algorithme de départ. Dans ces cas un autre algorithme doit être construit.

Plutôt que de recommencer à partir de rien, l'approche présentée ici permet de transformer un algorithme pour essayer d'en obtenir un autre qui donnera un programme plus efficace, cela tout en préservant sa correction. L'effort créatif pour transformer un algorithme est souvent moins important que pour le reconstruire entièrement.

Je me limiterai à la présentation des principes du système. Des exemples peuvent être trouvé dans [Deville 87]. Ce système est présenté pour permettre de mieux comprendre certaines remarques qui seront faites au chapitre 4.

1.2.4.1 Présentation : Règles d'inférence.

Le système utilise trois catégories de règles d'inférence :

A. Définition.

Définir un algorithme logique LA(p) tel que p n'apparaît dans aucune autre étape de la transformation.

B. Inférence logique.

Instanciation.

A partir de A <=> Def_A & C
inférer A & c <=> C & Def_A.

Dépliage (unfolding).

A partir de E <=> Def_E et F <=> Def_F,
où une sous-formule F' de Def_E est une instance de F,(F'=Fθ)
inférer E <=> New_def_E
où New_def_E est Def_E avec F' remplacé par Def_Fθ.

Pliage (folding).

A partir de E <=> Def_E et F<=> Def_F
où une sous-formule Def_F' de Def_E est une instance de Def_F
(Def_F'=Def_Fθ),
inférer E <=> New_def_E
où New_def_E est Def_E avec Def_F' remplacé par Fθ.

Groupement.

A partir de E & C1 <=> Def_1 et E & C2 <=> Def_2
inférer E & (C1 & C2) <=> Def_1 V Def_2.

C. Propriétés des spécifications.

Prenons quelques exemples:

A partir de la spécification de append on a que :
append(List_a,List_b,AB) & append(AB,list_c,list_abc) peut être remplacé par
append(List_b,List_c,BC) & append(List_a,BC,List_abc). Cela exprime l'associativité de append.

Si de la spécification de p(L) on sait que L est une liste,
de p(L) & (L=[] V ∃ H,T: L=[H|T] <=> Def
on peut inférer p(L) <=> Def.

1.3 PROGRAMME LOGIQUE.

Après avoir spécifié le problème à résoudre et avoir construit un algorithme logique pour celui-ci, nous allons voir comment dériver à partir de l'algorithme logique un programme logique dans un langage donné (PROLOG).

Cette dérivation se fait en deux temps. L'algorithme logique va d'abord être transformé en une procédure logique correcte, ensuite on examinera comment rendre cette première version plus performante en ajoutant des informations de contrôle.

1.3.1 Dérivation d'un programme logique correcte.

1.3.1.1 Règle de dérivation.

La première étape consiste à obtenir un ensemble de clauses à partir de l'algorithme logique à l'aide de règles de dérivation. Ces règles consistent à
-remplacer la condition suffisante et nécessaire <=>, par une condition nécessaire <=,
-éliminer les connecteurs V et les quantificateurs par des transformations basées sur des équivalences classiques en logique du premier ordre. [Lloyd 87], [Deville 87].
-remplacer <= par <- et les litéraux négatifs ¬B par not(B).

Exemple :

Je vais transformer la procédure member(E,L), construite dans la partie précédente et dont voici l'algorithme :

```
member(E,L) <=> L=[] & false
     V         L=[H¦T] & ( H=E
                         V   H=/=E & member(E,T)).
```

1) remplace <=> par <=

```
member(E,L) <= L=[] & false
     V         L=[H¦T] & ( H=E
                         V   H=/=E & member(E,T)).
```

2) par transformations successives j'obtiens :

```
member(E,L) <= L=[] & false
member(E,L) <= L=[H¦T] & ( H=E
                         V  ¬(H=E) & member(E,T)).

member(E,L) <= L=[H¦T] &  H=E
member(E,L) <= L=[H¦T] &  ¬(H=E) & member(E,T).
```

3) tranformation finale

```
member(E,L) <- L=[H¦T] &  H=E
member(E,L) <- L=[H¦T] &  not(H=E) & member(E,T).
```

1.3.1.2. Correction d'une procédure logique.

Etant donné une règle de calcul R, un programme P et une question Q, P U {Q} induit **un arbre de résolution, SLDNF-tree**. Chaque branche de l'arbre est une dérivation pour P U {Q}. Les branches qui correspondent à des dérivations correctes sont appelées **branches succès**. Une branche qui correspond à une dérivation infine est appelée une **branche infinie**.

Un **arbre de résolution fini** est un arbre qui ne contient pas de branche infinie.

Un arbre de résolution fini qui échoue est un arbre fini qui ne contient pas de branche succès.

A chaque branche succès correspond une **substitution réponse** θ pour P U {Q}.

Une règle de recherche est une stratégie pour chercher un arbre de résolution afin d'y trouver les branches succès.

La séquence de substitutions réponses pour P U {Q} est la séquence des substitutions réponses de P U {Q} atteintes selon la règle de recherche.

La correction d'une procédure logique est basée sur l'équivalence entre la relation décrite dans la spécification et la séquence de substitutions réponses, mais uniquement pour les termes qui respectent non seulement les préconditions de la spécification (types et autres préconditions) mais égalemnt la directionnalité et les préconditions de l'environnement.

Je vais maintenant présenter deux critères qui doivent être vérifiés pour qu'il y ait équivalence entre la relation et la séquence de substitutions réponses.

Soit :
- p la relation décrite dans la spécification de P.
- Subst, une séquence de substitutions réponses
- Sol l'ensemble des instances de base de Q $\in$ p et qui respectent les préconditions de P.

Subst est partiellement correct si toute instance de base de Q θ ($\theta \in$ Subst) qui respecte les préconditions est élément de Sol.

Subst est complet si $\forall$ élément S de Sol, $\exists$ θ Subst, tel que S est une instance de base de Q θ.

La méthode de construction garantit que :

- Subst est partiellement correct.
- Pour que Subst soit complet, il suffit de montrer que la règle de recherche est telle que chaque substitution réponse fait partie de Subst.

Pour qu'une procédure soit correcte il reste à montrer que :

- quand un litéral négatif est sélectionné, ses arguments sont toujours de base.
- L'arbre de résolution est fini quand la longueur de la séquence de substitutions réponses est finie.
- pour tout sous-problème qi, qui apparaît dans la procédure, toutes les préconditions de qi sont satisfaites quand qi est sélectionné:
  - types et autres préconditions,
  - directionnalité,
  - environnement.

1.3.1.2.1 Négation.

Pour des raisons d'efficacité, Prolog implémente la négation logique par la négation par échec. Cette façon de faire pose assez bien de problèmes au niveau logique.

La manière la plus simple de contourner ces problèmes est d'imposer que, avant la sélection d'un litéral négatif, ce litéral soit de base. Une manière simple est de considérer not comme une méta-procédure dont la directionnalité est In(gr) : Out(gr) <0-1>. De cette façon le problème de la négation sera résolu en même temps que la vérification de la directionnalité.

1.3.1.2.2 Directionnalité.

Le problème de vérification de la directionnalité d'une procédure est double:

-Etant donné une directionnalité en entrée, on doit vérifier que la directionnalité en sortie est bien respectée.
-La directionnalité est aussi une précondition à l'utilisation correcte d'une procédure, on doit donc vérifier que lors de la sélection d'un litéral par la règle de sélection de Prolog (sélection de gauche à droite) cette précondition est bien respectée.

La partie <min,max> concernant le nombre de substitutions réponses sera vue plus loin.

Vérification de la directionnalité : Analyse de flux.

Pour que la clause
p(X1,...,Xn) <- L1,...,Lm
soit correcte par rapport à la directionnalité
In(m_1,....m_n) Out(M1,...,Mn)
il faut que **V** j, la directionnalité de Lj soit vérifiée lors de sa sélection, c-à-d après l'exécution de L_{j-1} et que après Lm la partie Out de la directionnalité de p soit respectée.

Exemple :

Illustrons l'analyse de directionnalité avec la procédure append(L1,L2,L_app) que l'on a en partie spécifiée et construite dans la partie précédente. La dérivation de l'algorithme logique en une procédure logique donne la procédure suivante:

```
(1) append(L1,L2,L_app) <-
          L1=[],
          L_app=L2.
(2) append(L1,L2,L_app) <-
          L1=[H|T] ,
          L_app=[H|T_app],
          append(T,L2,T_app).
```

que l'on va vérifier pour la directionnalité:

In(gr,gr,any) Out(gr,gr,gr)

La notation mi ti Mi est utilisée pour décrire la forme de ti avant et après la sélection du litéral p(...,ti,...). Si ti n'est ni de base ni variable on utilise la notation f(...,mY Y MY,...) pour décrire que l'instanciation de ti

dépend de l'instanciation de Y. Ce cas est le plus délicat à traiter, il se peut que l'instanciation d'un terme ngv devienne gr parce que tous les termes qui le composent sont devenus gr. Cette possibilité est illustrée dans l'exemple.

Initialement (j=1), m_1 est la forme des variables Xi avant la sélection de L1. Une variable qui n'apparaît ni dans la tête ni dans L1,..L_{j-1} a la forme {var} quand Lj est sélectionné.

analysons la clause (2).

```
append( gr L1 gr ,gr L2 gr, any L_app gr ) <-
          gr L1 gr =[ var H gr ¦ var T gr] ,
          any L_app novar =[ gr H gr ¦ var T_app var],
          append(gr T gr , gr L2 gr, var T_app gr).
```

Les symboles en gras indiquent la forme finale des paramètres.

Il semble que L_app est novar après sa dernière occurrence, alors qu'il doit être gr. L_app est bien gr car T_app est devenu gr.

On remarque que l'on a dû permuter les deux derniers litéraux pour que l'analyse de flux réussisse.

1.3.1.2.3 Types.

Lorsque l'on a construit nos algorithmes on a toujours considéré que les paramètres étaient de base. En Prolog une procédure peut cependant être utilisée avec des variables ou des termes qui ne sont ni variable ni terme de base comme paramètres. Cela nous oblige à vérifier deux choses:

(1) pour tout paramètre actuel Ai qui respecte les préconditions, il existe une instance de base de Aiθ (θ une substitution réponse) qui respecte les préconditions.

(2) Avant la sélection de chaque litéral de la clause les préconditions sont respectées.

Nous nous limiterons ici à l'analyse des types, les autres préconditions peuvent être analysées de manière similaire.

La vérification des types peut se faire par une analyse de flux comme pour la vérification de la directionnalité, les deux problèmes étant similaires.

En cas de problème deux types de solutions sont possibles. Premièrement permuter les litéraux de la clause en faisant attention de ne pas détruire les résultats de la vérification de directionnalité (les deux peuvent se faire simultanément). Deuxièmement des litéraux qui forcent un terme à être d'un type donné peuvent être ajoutés. Cette solution va être illustrée par un exemple.

exemple.

procédure length_is_max(L).

Soit L une liste non vide d'entiers positifs.

Cette procédure détermine si

Le plus grand élément de L est la longueur de L.

directionnalité.
(1)In(gr) Out(gr)
(2)In(var) Out(gr).

Procédure logique.

lenth_is_max (L) <- length(L,X) , max_list(L,Z).

Spécifions également les sous-problèmes:

procédure length(L,N).

Soit L une liste,
N un entier positif ou nul.

Cette procédure détermine si

La longueur de L est N.

directionnalité.
In(any,any) Out(novar,gr)
In(gr,any) Out(gr,gr)

procédure Max_list(L,Max).

Soit L une liste non vide d'entiers,
Max un entier.

Cette procédure détermine si
Le plus grand élément de L est Max.

directionnalité.
In(any,any) Out(gr,gr)

Length_is_max est correct par rapport aux deux directionnalités. Regardons ce qui se passe avec les types.

(1) pour la première directionnalité :

-pas de problème pour le type de L car une liste non vide d'entiers positifs est aussi une liste et une liste d'entiers.
-pour le type de X il n'y a pas de problème avant length puisque X est une variable, ni avant max_list puique un entier positif ou nul est un entier.

(2) pour la seconde :

-avant length pas de problème puisque L est une variable, par contre avant max_list on sait juste que L est une liste or L doit être une liste non vide d'entiers. Il nous faut donc

ajouter un litéral Liste_non_vide_d_entiers(L) qui vérifie que L est bien une liste non vide d'entiers.
-pour X pas de problème.

Si on essaie de permuter les clauses on obtient :

length_is_max(L) <- max_list(L,X) & length(L,X)

qui est encore correct pour les deux directionnalités. Regardons pour les types.

Pour la seconde directionnalité :

-pas de problème pour le type de L,
-pour X il n'y a pas de problème avant max_list puisque X est une variable, par contre avant length il faut que X soit un entier postif ou nul or on sait juste que c'est un entier, il faut ajouter le litéral X>0 pour que la clause soit correcte.

1.3.1.3.4 Terminaison et complétude.

Nous allons maintenant traiter deux problèmes liés. Premièrement la terminaison de la procédure quand la séquence de substitutions réponses est finie. Deuxièmement, la complétude, qui se réduit à montrer que toute substitution réponse est atteinte tôt ou tard.

La terminaison et la complétude peuvent généralement être obtenues en permutant les litéraux à l'intérieur des clauses et en permutant les clauses.

On distingue deux cas, suivant que la séquence de substitutions réponses est infinie ou pas.

A) Séquence finie de substitutions réponses.

Dans ce cas-ci l'arbre de résolution, (SLDNF-tree), doit être fini pour satisfaire le critère de terminaison. Or si l'arbre de résolution est fini, toute solution réponse est atteinte tôt ou tard. Pour une question qui vérifie toutes les préconditions, si les arbres de résolution de tous les litéraux sélectionnés sont finis alors l'arbre de résolution de la question est fini.

A.1) Pas de récursivité et des arbres de résolution finis.

Dans l'analyse de flux on a choisi une directionnalité pour chaque litéral de la clause. On peut utiliser l'information quant au nombre de substitutions réponses <Min-Max> associées à la directionnalité. Si pour toutes clauses aucune des valeurs Max n'est infinie alors l'arbre de résolution est fini.

Cet argument n'est pas vrai pour les procédures récursives et étant donné que Max est une borne supérieure, une forme particulière des paramètres peut parfois induire un arbre fini même pour une valeur Max infinie.

A.2) Procédures récursives.

Pour les procédures récursives, chaque sous-problème (excepté la récursion) doit avoir un arbre de résolution fini. Une permutation des litéraux doit être effectuée quand ce n'est pas le cas. Pour les litéraux récursifs il nous faut une preuve par induction. Il faut prouver que pour n'importe quel paramètre ti, ti'<ti selon une relation bien fondée, avec ti' le paramètre correspondant à ti dans l'appel récursif. Pour les éléments minimaux l'arbre de résolution doit être fini.

L'induction ne peut être faite sur une variable car une relation bien fondée serait impossible à définir. On choisira donc un paramètre qui est de base dans la partie In de la directionnalité. Si le paramètre d'induction choisi lors de la construction de l'algorithme logique est de base, démontrer que la procédure se termine peut se limiter à démontrer que pour les cas minimaux l'arbre de résolution est fini. Typiquement on démontre que (la ou) les clauses qui traitent les cas minimaux ont toutes un arbre de résolution fini et que pour toutes les autres clauses l'arbre de résolution est un arbre fini qui échoue.

B) Séquence infinie de substitutions réponses.

Vu que la séquence de substitutions réponses doit être infinie cela n'a aucun sens de parler de terminaison. Cependant si on veut la complétude chaque substitution réponse doit éventuellement être atteinte.

Un seul sous-problème peut avoir un arbre infini car étant donné la règle de recherche de PROLOG si deux sous-problèmes ont un arbre infini après avoir atteint le deuxième sous-problème la recherche ne retournera jamais au premier.

S'il y a un arbre infini qui n'est pas dans la dernière clause, la complétude n'est obtenue que si les clauses suivantes ont toutes un arbre de recherche fini qui échoue puisque ces arbres ne seront jamais cherchés.

1.3.1.2.5 Vérification de <Min,Max>.

Si Min n'est pas respecté sa valeur doit être reconsidérée.

Si Max n'est pas respecté et qu'il est théoriquement possible de couvrir toutes les solutions avec une séquence de substitutions réponses de longueur Lg tel que Lg < Max alors l'algorithme doit être modifié sinon Max doit être modifié.

1.3.1.2.6 Effets secondaires.

Si un des sous-problèmes a des effets secondaires alors la procédure a des effets secondaires.

Le contrôle de ces effets secondaires ne peut être fait qu'avec une parfaite connaissance de la règle de calcul et de

recherche et spécialement du comportement des primitives avec effets secondaires lors d'un bactracking.

Si la spécification de la procédure est principalement la description d'effets secondaires, l'aspect déclaratif de la procédure n'est souvent pas très important et d'une manière générale on peut dire que la méthodologie ne s'applique pas bien à ce problème.

Si la relation est le point central de la spécification la méthodologie peut être appliquée et le problème des effets secondaires être analysé méticuleusement lors de la construction de la procédure logique.

1.3.2 Transformations de **procédure logique.**

Il est souvent plus facile de rendre efficace une procédure correcte que de corriger une procédure efficace.

Maintenant que nous avons obtenu une procédure logique correcte il ne nous reste plus qu'à la transformer pour la rendre plus efficace. Les transformations proposées ici sont basées sur la sémantique opérationnelle de PROLOG et l'introduction appropriée de primitives de contrôle.

Les conditions pour appliquer une transformation sont suffisantes mais pas toujours nécessaires.

Dans la suite P, Q et C représentent des litéraux positifs et S, T des séquences de litéraux.

Dans toutes les règles de transformation la nouvelle version sera séparée de l'ancienne par une barre horizontale.

Par exemple:

$$\frac{P \leftarrow S_1}{P \leftarrow S_1'}$$

1.3.2.1 Définition.

Un litéral p(t1,...,tn) est **déterministe** ssi la séquence de substitutions réponses pour ce litéral contient au plus une substitution réponse calculée.

Un litéral p(t1,...,tn) est **totalement déterministe** ssi la séquence de substitutions réponses pour ce litéral a une et seulement une substitution réponse calculée.

Un litéral p(t1,...,tn) est **infini** ssi la séquence de substitutions réponses est infinie.

La partie <Min,Max> de la directionnalité peut être utilisée pour détecter les litéraux (entièrement) déterministes ou infinis.

Le litéral p(t1,...tn) est **incompatible** avec le litéral q(s1,...,sm) ssi la séquence de substitutions réponses pour q(s1,...sm) est vide quand la séquence de substitutions réponses pour p(t1,...,tn) n'est pas vide.

Dans la suite, un cut (!) sera considéré comme un litéral, il est bien sûr entièrement déterministe.

1.3.2.2 Tranformations basées sur des arbres de résolution équivalents.

Quand deux clauses contiennent une séquence de litéraux identiques la transformation 1 permet d'éviter de construire deux arbres de résolution identiques pour cette séquence.

tranformation 1:

```
p(x1,...,xn) <- T, S1.
p(x1,...,xn) <- T, S2.
```

```
p(x1,...,xn) <- T, p1(y1,...,ym).
p1(y1,...,ym) <- S1.
p1(y1,...,ym) <- S2.
```

où -y1,...,Ym est l'ensemble des variables de S1 et S2.
-p n'a pas d'effet secondaire.
-T,S1,S2 ne contiennent pas de cut,
-p n'est pas infini.

La transformation 2 est basée sur la règle de négation par échecs et permet également d'éviter de construire deux arbres de résolution redondants.

transformation 2:

```
p(x1,...,xn) <- T, C, S1.
p(x1,...,xn) <- T, not(C), S2.
```

```
p(x1,...,xn) <- T, C, !, S1.
p(x1,...,xn) <- T, S2.
```

où - T, C n'ont pas d'effet secondaire.
- T est déterministe.

La transformation suivante est basée sur l'existence de litéraux incompatibles dans les différentes clauses. Elle est particulièrement intéressante pour des procédures construites par induction structurelle. Dans ce cas les C1 qui déterminent la forme stucturelle du paramètre d'induction sont souvent incompatibles.

transformation 3:

```
p(x1,...,xn) <- C1, S1.
p(x1,...,xn) <- C2, S2.
    .                .
p(x1,...,xn) <- Cm-1, Sm-1.
p(x1,...,xn) <- Cm, Sm
```

```
p(x1,...,xn) <- C1, !, S1.
p(x1,...,xn) <- C2, !, S2.
    .                .
p(x1,...,xn) <- Cm-1, !, Sm-1.
p(x1,...,xn) <- Cm, Sm
```

où -les Ci sont déterministes
-les Ci n'ont pas d'effet secondaire
-Ci est imcompatible avec Cj (i=/=j).

1.3.2.3 Transformations basées sur les règles de calcul et de recherche.

Quand un litéral est déterministe il est possible de réduire l'arbre de recherche en évitant le backtracking par un cut. Les deux transformations suivantes sont basées sur ce principe.

transformation 4:

```
p(x1,...,xn) <- S1
    .              .
p(x1,...,xn) <- Sm1, Sm2
```

```
p(x1,...,xn) <- S1
    .              .
p(x1,...,xn) <- Sm1, !, Sm2
```

où -Sm1 n'a pas d'effet secondaire
-Sm1 est déterministe.

transformation 5:

```
p(x1,...,xn) <- S1
     .                .
     .                .
p(x1,...,xn) <- Si1, !, Si2, Si3
     .                .
     .                .
p(x1,...,xn) <- Sm.
```

```
p(x1,...,xn) <- S1
     .                .
     .                .
p(x1,...,xn) <- Si1, !, Si2, !, Si3
     .                .
     .                .
p(x1,...,xn) <- Sm.
```

où -Si2 est déterministe,
-Si2 n'a pas d'effet secondaire.

1.3.2.4 Transformations basées sur une évaluation partielle.

La transformation 6 réalise une évaluation partielle en ouvrant certains litéraux. Cette transformation est semblable à la règle de dépliage du système de tranformation d'algorithmes logiques.

transformation 6:

```
p(x1,...,xn) <- S1
     .                .
     .                .
p(x1,...,xn) <- Si1, q(t1,...,tk), Si2.
     .                .
     .                .
p(x1,...,xn) <- Sm.
q(y1,...,yk) <- T.
```

```
p(x1,...,xn) <- S1
     .                .
     .                .
p(x1,...,xn) <- Si1, t1=y1, ..., tk=yk, T, Si2.
     .                .
     .                .
p(x1,...,xn) <- Sm.
q(y1,...,yk) <- T.
```

où -il n'y a pas de variable commune entre y1, ..., yk, T et x1, ..., xn, t1, ..., tk, Si1, Si2.
-T ne contient pas de cut.

1.3.2.5 Transformations basées sur l'égalité.

La tête de nos clauses ne contient que des variables ce qui est contraire à l'habitude de la plupart des programmeurs Prolog qui instancient la tête de la clause autant que possible. Les transformations que nous allons présenter maintenant permettent d'instancier la tête de la clause.

La transformation 7 permet de propager l'égalité vers la fin de la clause.

tranformation 7:

```
p(x1,...,xn) <- S1, Y=t, S2.
```

```
p(x1,...,xn) <- S1, Y=t, S2{t/Y}.
```

En particularisant 7 quand S1 est vide on obtient 8 qui permet de transformer aussi bien la tête que la queue de la clause.

transformation 8:

```
p(x1,...,xn) <- Y=t, S.
```

```
p(x1,...,xn){t/Y} <- Y=t, S{t/Y}.
```

Dans certains cas il est possible de supprimer un litéral égalité.

transformation 9:

```
p(x1,...,xn) <- S1, Y=t, S2.
```

```
p(x1,...,xn) <- S1, S2.
```

où Y n'apparaît pas dans x1, ..., xn, S1, S2 et t

Quand l'égalité est après un cut la transformation 10 nous permet de transformer la clause pour pouvoir utiliser la 8.

transformation 10:

```
p(x1,...,xn) <- S1, !, q(y1,...,ym), S2.
```

```
p(x1,...,xn) <- S1, q(y1,...,ym), !, S2.
```

où - q n'a pas d'effet secondaire,
- q(y1,...,ym) est entièrement déterministe.

1.3.2.6 Transformations basées sur la récursion terminale.

Une procédure logique est "**tail recurcive**" ssi elle possède un et un seul sous-but récursif et sa dernière clause a la forme:

p(x1,...,xn) <- S, p(y1,...,yn)

où S est déterministe.

Quand la dernière clause a cette forme mais qu'il y a plus d'un sous-but récursif, la procédure est appelée "**semi tail recurcive**".

La définition d'une procédure "tail recurcive" peut être généralisée aux cas où la clause récursive n'est pas la dernière.

La récursion terminale permet un gain aussi bien en temps d'exécution qu'en place mémoire utilisée.

Quand plusieurs permutations des litéraux sont correctes par rapport aux spécifications, celle qui permet d'obtenir une procédure "tail recurcive" est préférée aux autres. Si aucune permutation ne permet d'obtenir une procédure "tail récursive" une tranformation ou une généralisation de l'algorithme logique permet parfois d'obtenir une procédure "tail recurcive".

Une procédure "tail recurcive" peut être assimilée à une procédure itérative. Certains Prolog ne sont pas capables de reconnaître si S est déterministe. Le moyen le plus simple pour que le système le reconnaisse est d'appliquer la transformation 4 pour insérer un cut après S.

1.3.2.7 Transformations basées sur les techniques d'implémentation de Prolog.

L'utilisation de la variable anonyme permet un gain d'efficacité dans certaines implémentations de Prolog.

transformation 11:

p(x1,...,xn) <- S.
_____________________u_

p(x1,...,xn){ _/Y} <- S{ _/Y}

où Y apparaît seulement une fois dans x1, ..., xn ou dans S (mais pas dans les deux).

Dans la suite du mémoire on parlera des déclarations de mode qui sont une autre technique d'optimisation basée sur les implémentations de Prolog. On peut également citer l'indexage de clauses.

1.4 CONCLUSION.

La méthodologie qui vient d'être présentée va servir à construire une application. Dans le chapitre 5, la manière dont la méthodologie a été utilisée ainsi que quelques extensions et modifications seront présentées.

Chapitre 2

APPLICATION TEST

Afin d'évaluer la méthodologie présentée au chapitre 1 j'ai écrit un programme pour générer des déclarations de mode dans le cadre de recherches faites à la K.U.Leuven sur la compilation d'informations de contrôle.

Pour permettre au lecteur de situer le programme réalisé je vais d'abord présenter la compilation d'informations de contrôle. Je présenterai ensuite comment générer des déclarations de mode. Ces deux parties sont indépendantes l'une de l'autre, des définitions et notations introduites dans la première partie sont néanmoins utilisées dans la seconde.

2.1 Compilation d'informations de contrôle.

Deux facteurs déterminent si un programme est efficace ou pas. La manière dont le programme est écrit, partie logique, et la manière dont il est exécuté, partie contrôle. En programmation logique cela est souvent exprimé par la pseudo-équation de Kowalski: Algorithme = Logique + contrôle [Kowalski 79]. De nombreuses recherches en programmation logique ont pour but d'essayer de séparer ces deux composants. Par exemple dans la méthodologie présentée au chapitre précédent on construit d'abord un algorithme logique sans se soucier de l'exécution de celui-ci et dans une seconde phase on le transforme en une procédure logique en tenant compte des mécanismes d'exécution de Prolog.

En Prolog la règle de calcul est fixe. Une règle fixe étant rarement bonne dans tous les cas, il arrive que certains algorithmes, élégants d'un point de vue déclaratif doivent être transformés en une version plus obscure car ils sont trop inefficaces (ex: tri par permutation). Dans d'autres cas, il n'est pas possible d'obtenir un algorithme vraiment efficace (ex: 8-Queens).

Deux solutions à ce problème ont été proposées: écrire un meta-interprèteur qui permette d'obtenir l'exécution souhaitée, ou créer de nouveaux langages qui permettent de définir la règle de calcul désirée (ex: IC-Prolog, mu-Prolog).

La méthode que nous allons présenter [Bruynooghe86, De Schreye 87] permet de transformer un programme auquel est associée une règle de calcul spécifique en un programme Prolog (programme exécuté avec la règle de calcul de prolog) équivalent.

Grâce à cette méthode la règle de calcul est en quelque sorte compilée dans le programme Prolog ce qui permet d'économiser le temps utilisé par le meta-interprèteur pour obtenir la règle de calcul souhaitée lors de l'exécution du programme.

Après quelques rappels, je présenterai quel type d'informations de contrôle le système permet de compiler et comment cette compilation est réalisée. Le but n'est pas de donner une présentation précise et technique du système mais plutôt de permettre au lecteur de comprendre son fonctionnement et de voir comment l'information présente permet de générer des déclarations de mode pour les programmes "compilés".

2.1.1 Rappel.

Une **règle de calcul** est une fonction d'un ensemble de questions vers un ensemble d'atomes, telle que la valeur de la fonction pour une question est un atome de cette question.

Soit une question G, <- A1,...,Am,...,Ak, et une clause C, A <- B1,...,Bq. G' est la **question dérivée** à partir de G et C en utilisant la substitution θ si:
-Am est l'atome sélectionné dans G par la règle de calcul,
-θ est le mgu de Am et A,
-G' est la question <-(A1,....,A_{m-1},B1,...,Bq,A_{m+1},...Ak)θ.

Soit P un programme, G une question. **Une dérivation de P U {G}** consiste en une séquence G_0 = G, G1, ... de questions, une séquence C1, C2,.. de variantes de clauses de P et une séquence θ1, θ2,... de substitutions telle que G_{i+1} est dérivée à partir de Gi et C_{i+1} en utilisant θ_{i+1}.

2.1.2 Règle de calcul et mode de calcul.

2.1.2.1 Règle de calcul basée sur l'instanciation des sous-buts.

La règle de calcul que nous allons présenter est comme son nom l'indique basée sur l'instanciation des sous-buts de la question.

Pour pouvoir définir cette règle de calcul, nous avons besoin d'une notation pour exprimer l'instanciation d'un sous-but selon un certain degré d'abstraction:

Soit A_0={g, v, any}. Un **terme d'instanciation_0** est soit un élément de A_0 soit une expression de la forme f(t1,...tn), où t1,...tn sont des termes d'instanciation_0, f un foncteur n-aire et n≥1.

Un terme d'instanciation_0, T0, décrit l'instanciation d'un terme, T, si
T0 = g et T est un terme de base,
T0 = v et T est une variable,
T0 = f(t1,...,tn) et T = f(u1,....,un) et **V** i: 0≤i≤n, ti décrit l'instanciation de ui.
T0= any et T est un terme.

Remarque: any est utilisé pour des termes dont l'instanciation n'a pas d'importance pour la règle de calcul.

Un **atome d'instanciation_0**, noté atome_0, est une expression de la forme p(t1,...tn) avec t1,...,tn des termes d'instanciation_0, p un symbole de prédicat n-aire et n≥0.

Un atome d'instanciation_0, A, décrit l'instanciation d'un prédicat, P, si
A = p(t1,...,tn) et P=p(u1,....,un), n≥0, et **V** i: 0≤i≤n, ti décrit l'instanciation de ui.

exemple:

Avec la convention que les variables commencent par une majuscule et les constantes par une minuscule, name(g,v,any) décrit l'instanciation de name(toto,X,tutu) ou de name(12,Toto,Y) mais pas de name(X,tutu,toto).

Une règle de calcul basée sur l'instanciation des sous-buts est définie par un ensemble de couples (S,s) où S est un ensemble fini d'atomes_0, et s ∈ S.
Soit une question <-q_1,...q_n, un sous-but q_i est sélectionné par la règle de calcul de la manière suivante:
(1)choisir un couple (S_0,s_0) tel que
- **V** q_j **∃** A ∈ S_0 tel que A décrit l'instanciation de q_j.
- **∃** q_j tel que s_0 décrit l'instanciation de q_j.

(2)parmi les q_j dont l'instanciation est décrite par s_0 choisir q_i selon une règle fixe, e.g. l'atome qui a le plus récemment obtenu l'instanciation décrite par s.

remarque:

Des contraintes supplémentaires sont imposées pour que la règle de calcul définie soit complète, cohérente, bien définie. Nous ne parlerons toutefois pas de ces contraintes ici car elles ne sont pas indispensables pour comprendre le reste de l'exposé.

2.1.2.2 Mode de calcul.

Lors de la compilation d'un programme on désire ne pas transformer certaines procédures qui ont déjà une exécution efficace avec la règle de calcul de prolog. Pour cette raison, on introduit la notion de mode de calcul.

Le **mode de calcul** détermine si un sous-but doit être entièrement résolu (mode solve) lorsqu'il est sélectionné par la règle de calcul, ou si un seul pas d'inférence est effectué (mode expand). Les sous-buts qui doivent être entièrement résolus le sont avec la règle de calcul de prolog.

Le mode de calcul est défini en ajoutant à chaque couple (S,s) de la règle de calcul, l'indication 'solve' ou 'expand' suivant que le sous-but sélectionné doit être entièrement résolu ou qu'un seul pas d'inférence doit être effectué. On obtient ainsi un ensemble de triples (S,s,c).

remarque :

dans la suite, par règle de calcul nous nous référerons à l'ensemble des triples (S,s,c).

exemple:

une règle de calcul pour le programme de tri par permutation:

```
sort(X, Y) :- perm(X, Y) , ord(Y).
perm([], []).
perm([X| Y], [U| V]) :- delete(U, [X| Y], W),perm(W, V).
delete(X, [X| Y], Y).
delete(X, [Y| U], [Y| V]) :- delete(X, U, V).
ord([]).
ord([X]).
ord([X,Y| Z]) :- X ≤ Y, ord(Y, Z).
```

qui permette d'entrelacer perm et ord pour des questions dont l'instanciation est définie par l'ensemble des triples,

```
{ ({sort(g,v)},sort(g,v),expand),
  ({perm(g,v),ord(v)},perm(g,v)expand),
  ({ord(g)},ord(g),expand)
  ({delete(v,g,v),perm(v,v),ord([v|v])},delete(v,g,v),solve)
  ({perm(g,v),ord([g|v])},perm(g,v),expand)
  ({del(v,g,v),perm(v,v),ord([g,g|v])},del(v,g,v),solve)
  ({perm(g,v),ord([g,g|v])},ord([g,g|v]),expand)
  ({g≤g,perm(g,v),ord([g|v])},g≤g,solve)                    }
```

2.1.3 Méthode de compilation.

Etant donné un programme P, un atome_0 q, et une règle de calcul basée sur l'instanciation des sous-buts, **C(P,q)**, on désire obtenir ('compiler') un programme P' qui, exécuté en prolog, donne pour toute question dont l'instanciation est décrite par q une réponse identique à P exécuté selon C(P,q).

Le programme P auquel on ne pose que des questions dont l'instanciation est décrite par l'atome_0 q, sera **noté (P,q)**.

La méthode de compilation se base sur l'arbre de résolution d'une exécution symbolique de (P,q) selon C(P,q). Après avoir définit (a) de nouvelles notations pour décrire l'instanciation d'un ensemble de prédicats , je définirai (b) ce qu'est un arbre de résolution symbolique. J'expliquerai ensuite (c) comment un programme Prolog peut-être obtenu.

2.1.3.1 Définitions, notations.

Dans la suite on désire exprimer l'instanciation des différents prédicats selon une granularité plus fine que lors de la définition de la règle de calcul. On veut pouvoir exprimer les occurrences multiples d'un même terme. On définit pour cela les notations suivantes.

Soit A_1 ={constantes qui apparaissent dans le programme compilé} U {variables Vi| i ∈ N} U {variables Gi| i ∈ N}

Un **terme d'instanciation_1** est soit un élément de A_1 soit une expression de la forme f(t1,...tn), où t1,...tn sont des termes d'instanciation_1, f est un foncteur n-aire et n≥1.

Fv(X)=n est une fonction de l'ensemble des variables dans l'ensemble des entiers tel que Fv(X)=Fv(Y) ssi X=Y.

Fg(x) est une fonction de l'ensemble des termes de base dans l'ensemble des entiers tel que Fg(x)=Fg(y) ssi x=y.

Un terme d'instanciation_1, T1, décrit l'instanciation d'un terme, T, si

T1 = constante et T est cette constante,
T1 = Gi et T est un terme de base, et Fg(T)=i,
T1 = Vi et T est une variable, et Fv(T)=i,
T1 = f(t1,...,tn) et T = f(u1,....,un) et **V** i: 0≤i≤n, ti décrit l'instanciation de ui.

Un **atome d'instanciation_1**, noté atome_1, est une expression de la forme p(t1,...tn) avec t1,...,tn des termes d'instanciation_1, p un symbole de prédicat n-aire et n≥0.

Un atome d'instanciation_1, A, décrit l'instanciation d'un prédicat, P, si

A = p(t1,...,tn) et P=p(u1,....,un), n≥0, et **V** i: 0≤i≤n, ti décrit l'instanciation de ui.

F1(P)=A1 est une fonction d'un prédicat P vers un atome_1 A1 tel que A1 décrit l'instanciation du prédicat P.

exemple:

Avec la convention que les variables commencent par une majuscule et les constantes par une minuscule, name(G1,V3,G1) décrit l'instanciation de name(toto,X,toto) mais pas de name(toto,X,tutu) ou de name(X,toto,X).

La construction d'un arbre de résolution symbolique peut être plus facilement expliquée par unification d'atomes_1. L'unification de variables_1 est un peu différente de l'unification de variables prolog, en voici les règles:

Unification de deux termes d'instanciation_1:

-Vi s'unifie avec Vj et θ={ Vi/Vj }
-Vi s'unifie avec Gj et θ={ Vi/gj }
-Vi s'unifie avec f(....) et θ={ Vi/f(...) }
-Vi s'unifie avec const et θ={ Vi/const }
-Gi s'unifie avec Vj et θ={ Vj/Gi }
-Gi s'unifie avec Gj et θ={ Gi/Gj }
-Gi s'unifie avec f(...,Vj,...) et θ={Gi/f(...,Vj,...),Vj/Gk }
-Gi s'unifie avec const et θ={ Gi/const }
-f(t1,...,tn) s'unifie à f(u1,...,un) si ∀ i, ti s'unifie à ui.
-deux constantes s'unifient si elles sont identiques.

On définit également les fonctions suivantes qui sont utiles pour faire correspondre un atome_1 à un atome_0 :

F'(E1)=E0 est une fonction qui transforme un ensemble d'atomes_1 en un ensemble d'atomes_0 comme suit: chaque occurrence d'une constante ou d'une variable Gi est remplacée par g, d'une variable Vi par v. Les termes de la forme f(g,g,..,g) sont remplacés par g.

Fi''(E0)=E0', $1 \leq i \leq n$, est un ensemble de fonctions qui transforment un atome_0 en un atome_0 en remplaçant certains termes par any.

Fi(E1)=E0 est la composition de F' et de Fi''.

Une **règle de calcul est représentée de manière unique** si dans un triple (S,s,c) de la règle de calcul, un atome_0 $\in$ S contient any en une certaine position alors dans tous les triples (S',s',c') les atomes_0 $\in$ S' avec le même symbole de prédicat ont any dans cette position.

Si on impose à la règle de calcul d'être représentée de manière unique, alors il suffit d'une fonction Fi pour faire correspondre un ensemble d'atomes_1 à un ensemble d'atomes_0. Soit F_∞ cette fonction.

Deux atomes_1, A, B, sont de même type si $F_\infty(A) = F_\infty(B)$.

2.1.3.2 Arbre de résolution symbolique .

Un **arbre de résolution symbolique** pour un programme (P,q) avec la règle de calcul C(P,q) est un arbre dont:

-la racine est q,

-chaque noeud est un ensemble d'atomes_1.

-Soit T={A1,...,Am,...Ak} (k≥1) un noeud de l'arbre, un triple (S,s,c), de la règle de calcul, est choisi tel que F⊙(T)=S, puis parmi les Aj tel que F⊙(Aj)=s, un atome Ai est choisi selon une règle fixe.

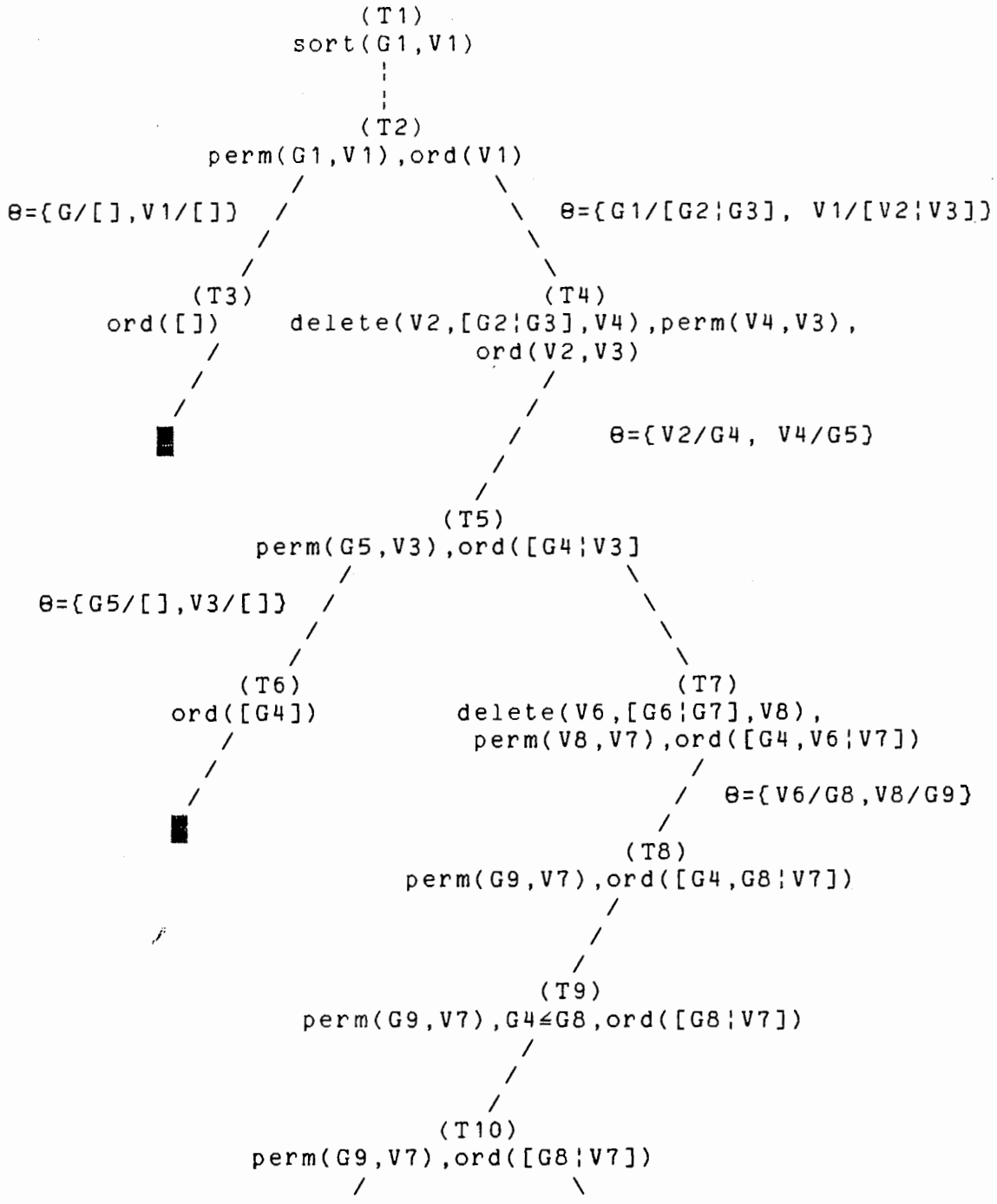
  -si le mode de calcul dit que Ai doit être entièrement résolu (c=solve), il faut exprimer l'effet de résoudre complètement Ai sur les Vj qui apparaissent dans Ai. S'il y a n substitutions $\theta_1,\ldots,\theta_n$, possibles, le noeud T aura n fils $T_p=\{(A1,\ldots A_{i-1},A_i,\ldots,Ak)\theta_p\}$, $1\leq p\leq n$

  -si le mode de calcul dit qu'un seul pas d'inférence doit être fait (c=expand), alors pour chaque clause F1(Cl) = A <- B1,...,Bq avec Cl une clause de P, telle que A et Ai s'unifient avec mgu θ, il existe un fils $T'=\{(A1,\ldots,A_{i-1},B1,\ldots,Bq,A_{i+1},\ldots,Ak)\theta\}$.

-chaque arc est étiqueté avec les substitutions nécessaires, θ, pour passer du père au fils.

exemple :

l'arbre de résolution symbolique pour le programme de tri par permutation pour des questions dont l'instanciation est décrite par sort(G1,V1), avec la règle de calcul présentée précédemment est :



remarque :

l'arbre de résolution pour une question dont l'instanciation est décrite par sort(G1,V1), est une instance d'un sous-arbre de l'arbre de résolution symbolique.

2.1.3.3 Synthèse d'un programme Prolog.

A partir de l'arbre de résolution symbolique, il est possible sans grand problème d'obtenir un programme prolog infini, dont l'arbre de résolution symbolique est équivalent, en introduisant un prédicat différent pour chaque noeud et une clause pour chaque arc.

Pour obtenir un programme fini, on va regrouper les noeuds en un nombre fini de classes de noeuds similaires(a), et les arcs en classes d'arcs similaires(b). Cela revient en quelque sorte à "replier" l'arbre de résolution symbolique en un graphe qui ne contient plus qu'un nombre fini de noeuds et d'arcs. Le programme proprement dit est ensuite obtenu en introduisant un prédicat(c) pour chaque classe de noeuds similaires et une clause(d) pour chaque classe d'arcs similaires.

a. Noeuds similaires.

On définit la similarité entre deux noeuds comme suit:

Deux noeuds, Ti, Tj, sont similaires, si le même type de sous-buts est sélectionné de Ti et de Tj par la règle de calcul.

Exemple:

Dans l'arbre de résolution pour le programme de tri, T10 est similaire à T5 car dans les deux cas un sous-but de type perm(g,v) est sélectionné.

b. Arcs similaires.

Soit un arc d'un noeud Ti à un noeud Tj, Xij un entier qui identifie la clause utilisée pour passer de Ti à Tj, ou 0 si le sous-but sélectionné est entièrement résolu et θij les substitutions pour passer de Ti à Tj. Le tuple (Ti,Tj,Xij,θij) définit cet arc.

Deux arcs (Ti,Tj,Xij,θij) et (Ti',tj',Xij',$\theta ij'$) sont similaires si:
- Ti et Ti' sont similaires,
- Tj et Tj' sont similaires,
- Xij=Xij'.

c. Génération des prédicats.

Nous allons maintenant voir comment il est possible d'obtenir un nouveau prédicat pour chaque classe de noeuds similaires.

En fait nous allons définir une fonction New(T), d'un noeud T $\in$ une classe de noeuds similaires C vers une occurrence du prédicat défini pour C.

Newname(C) est une fonction qui associe un symbole de prédicat unique à toute classe de noeuds similaires.

$Fo^*(\mathbb{C}) = \bigcup_{T \in \mathbb{C}} Fo(T)$ est une fonction d'un ensemble de noeuds similaires C vers un ensemble d'atomes_0 T tel que pour chaque atome_1 élément d'un noeud de C, il existe un atome_0 $\in$ T qui décrive cet atome_1. On a que T contient autant d'atomes_0 que d'atomes_1 de types différents dans les noeuds de C.

Soit $Fo^*(C)=\{s_1,\ldots,s_m\}$ et $(s_{i1},\ldots,s_{in})$ l'ordre lexicographique de $Fo^*(C)$, **New(T)=Newname(C)(L_1,...,L_m)** avec T $\in$ C et Lj est la liste $[t_1,\ldots,t_m]$ de tous les sous-buts $t_k \in T$ tel que $Fo(t_k)=s_{ij}$.

On remarque que $\bigcup_{j=1,n} \{ t \mid t \in L_j \} = T$

d. Génération des clauses.

Finalement, on obtient le programme en introduisant une clause pour chaque classe d'arcs similaires. Deux types de clauses différents sont utilisés suivant que le mode de calcul est solve ou expand.

Soit $C=\{ (I_j,F_j,N_j,\theta_j) \mid 1 \leq j \leq k \}$ un ensemble d'arcs similaires. Par définition il existe deux ensembles de noeuds similaires C1, C2 tel que **∀** j, $I_j \in C1$ et $F_j \in C2$.

De la définition d'une classe d'arcs similaires et de New(C), il découle que le sous-but sélectionné (noté Sb) appartient toujours à la même liste, **∃** i: Sb $\in$ Li. On peut supposer en toute généralité que Sb est le premier élément de Li.

d.1 Mode de calcul solve.

Avec le mode de calcul solve la clause a la forme suivante:

```
Newname(C1)(L1,...,Li-1,[Hi|Ti],Li+1,...,Ln1) <-
     Hi,
     Shuffle=(L1,...,Li-1,Ti,Li+1,...,Ln1, L1',...,Ln2'),
     Newname(C2)(L1',...,Ln2').
```

Cette clause est telle que **∀** j, sa tête s'unifie à New(Ij), avec mgu θ. De plus, sa structure est telle que New(Ij)θ=New(Ij) et que (Hi)θ = Sb.

Après unification de New(Ij) et de la tête de la clause, on a que $\bigcup \{ t \mid t \in L_1,\ldots,L_{i-1},[H_i|T_i],L_{i+1},\ldots,L_{n1} \} = I_j$.

Après résolution de (Hi)θ=Sb on a que :

$\bigcup \{ t \mid t \in L_1,\ldots,L_{i-1},T_i,L_{i+1},\ldots,L_{n1} \} = F_j$.

Le but de Shuffle= est de répartir les éléments des listes L1,...Ln1 dans les listes L1',...Ln2' afin qu'après Shuffle= on obtiennent New(Fj)=Newname(C2)(L1',...,Ln2').

d.2 Mode de calcul expand.

Avec le mode de calcul expand une clause de l'ancien programme est utilisée pour passer au noeud suivant. Soit h la tête de cette clause et b la queue. θ est le mgu de h et de la plus faible anti-occurrence commune (least common anti-instance, lca) de Sb, lca(Sb). On obtient une clause de la forme :

Newname(C1)($L_1,\ldots,L_{i-1},[h\ \theta\ |T_1],L_{i+1},\ldots,Ln_1$) <-
Shuffle$_\alpha$($L_1,\ldots,L_{i-1},T_1,L_{i+1},\ldots,Lm_1$,b θ $L_1',\ldots,Lm_2'$),
Newname(C2)($L_1',\ldots,Lm_2'$).

Cette clause est telle que $\forall$ j, sa tête s'unifie à New(Ij), avec le mgu Ω.

Après l'unification de New(Ij) et de la tête on a que :

$U\ \{t\,|\,t \in L_1,\ldots,L_{i-1},T_1,L_{i+1},\ldots,Ln_1,b\ \theta\} = Fj.$

Le but de Shuffle$_\alpha$ est le même que précédemment

Remarque :

Dans le cas où il existe des transitions d'un ensemble de noeuds similaires dans plusieurs ensembles de noeuds similaires différents, il peut arriver que la tête d'une clause ne soit pas assez discriminante et que le nouveau programme permette d'obtenir des noeuds qui n'existent pas dans l'arbre du programme original. Dans ce cas, un prédicat est ajouté dans la clause pour éviter ce problème.

2.1.3.4 Remarques.

L'arbre de résolution symbolique de départ est souvent infini. Pour pouvoir faire la transformation, on se base sur un sous-arbre fini de celui-ci. On suppose que ce sous-arbre est suffisamment grand pour pouvoir faire correctement la transformation. Il est possible de démontrer par induction si ce sous-arbre est suffisamment grand, cette démonstration ne semble malheureusement pas facilement automatisable.

Il est également souvent possible de réduire la taille l'arbre de départ (in line expansion).En gros, les noeuds qui n'ont qu'un fils sont supprimés et l'arc qui arrive et l'arc qui part de ce noeud sont remplacés par un nouvel arc du père vers le fils du noeud. Cela permet de réduire la taille du programme compilé.

L'équivalence entre le programme de départ et le programme compilé est démontrée par équivalence de leurs arbres de résolution.

2.2 Déclaration de mode.

Après avoir rappelé ce qu'est une déclaration de mode, je montrerai comment généraliser l'information quant à l'instanciation des prédicats, puis comment transformer le programme pour pouvoir générer des déclarations de mode.

2.2.1 Rappel.

Une **déclaration de mode** est une directive de compilation qui indique au compilateur l'instanciation des arguments d'un prédicat lors de sa sélection par la règle de calcul, (au moment de l'appel). Une déclaration de mode pour un prédicat P a la forme suivante: P(...,O,...,I,...,?,...). O indique qu'un argument est une variable (Output), I qu'un argument est un terme de base (Input), ? est utilisé pour les arguments dont on ne connaît pas l'instanciation ou qui ne sont ni variables ni termes de base.

2.2.2 Forme générale de l'instanciation d'un prédicat.

Soit une clause :

$p(t_1,...,t_n)$ <-
...,
$q(l_1,...,l_m)$.

et A1,...,Ak des atomes_1 qui décrivent les k instanciations possibles du prédicat p/n au moment de sa sélection par la règle de calcul.

Pour pouvoir générer une déclaration de mode pour P/n, on a besoin de l'instanciation de p/n lors de n'importe quel appel. Nous allons voir comment obtenir cette information après avoir défini de nouvelles notations.

Un **atome d'instanciation_2**, noté atome_2, est défini de la même manière qu'un atome_1 sauf que l'ensemble de départ
A2 = A1 U {variable Ni ¦ i un entier positif}.

Une variable Ni indique que l'on ne connaît pas l'instanciation d'un terme (any). Etant donné la définition d'un atome_1 et d'un atome_2, on remarque qu'un atome_1 est un cas particulier d'un atome_2.

Voyons maintenant comment on peut obtenir un atome_2 qui décrive l'instanciation commune à plusieurs atomes_2.

Soit Fct une fonction injective de N^2->N, **un terme_2 Tg est la généralisation de deux termes_2** T1 et T2 en fonction de Fct ssi
- $Tg=V_k$ quand $T1=V_i$ et $T2=V_j$ avec k=Fct(i,j).
- $Tg=G_i$ quand $Fo(T1)=G_i$ et $Fo(T2)=g$
- $Tg=f(t_1,...,t_n)$ quand $T1=f(t_{11},...,T_{1n})$ et $T2=f(t_{21},...,t_{2n})$ avec $ti=Gen(t_{1i},t_{2i})$ **V** i: $1 \leq i \leq n$, et si Tg contient deux variables N_i et N_j ou deux variables G_i et G_j alors i=/=j.
- $Tg=N_i$ sinon.

Un terme_2 Tg est la généralisation de deux termes_2 T1 et T2, **noté Tg=gen(T1,T2)** s'il existe une fonction injective Fct, telle que Tg est la généralisation de T1 et T2 en fonction de Fct.

Quand on généralise deux atomes_2, on est intéressé par les informations suivantes : savoir si un terme est une variable ou un terme de base dans les deux atomes_2, ou si l'instanciation de ce terme varie de l'un à l'autre. On est également intéressé par les liens entre variables_2 Vi mais pas entre les variables_2 Gi et Ni. Les termes de base ou les termes dont l'instantiation n'est pas la même dans les deux atomes sont donc décrits par des variables Gi et Ni différentes pour indiquer qu'ils ne sont pas liés.

Un atome_2, Ag est la généralisation de deux atomes_2, A1 et A2, noté Ag=Gen(A1,A2) si
-Ag=p(t_1,...,t_n) quand A1=p(t_{11},...,t_{1n}) et A2=p(t_{21},...,t_{2n})
avec ti=Gen(t_{1i},t_{2i}) **V** i: $1 \leq i \leq n$, et si Ag contient deux variables Ni et Nj ou deux variables G_i et G_j alors i=/=j.
-Ag=Ni sinon.

Ag est la généralisation d'un ensemble non vide d'atomes_2, C={A1,...An},noté Ag=Gen(C), si Ag=Gen(A1,Gen(...,Gen(An-1,An))) quand $n \geq 2$ et Ag=A1 sinon.

remarque:

Pour les programmes obtenus par compilation d'informations de contrôle, l'instanciation des prédicats au moment de l'appel peut facilement être obtenue. Soit T1,...Tn les différents noeuds de l'arbre de résolution symbolique à partir duquel le programme a été compilé, les atomes_2 New(Ti) ,$1 \leq i \leq n$, décrivent les instanciations des nouveaux prédicats lors de leur sélection. L'arbre de résolution symbolique contient également l'instanciation des prédicats qui sont sélectionnés pour être entièrement résolus et qui apparaissent dans le nouveau programme.

2.2.3 Transformation du programme et déclaration de mode.

On vient de voir, comment on peut obtenir l'instanciation générale de certains prédicats du nouveau programme lorsqu'ils sont sélectionnés par la règle de calcul, ce qui devrait nous permettre de faire facilement une déclaration de mode pour ceux-ci. Malheureusement, une déclaration de mode ne nous permet que de déclarer un argument comme étant une variable, un terme de base ou inconnu. Déclarer qu'on ne sait rien à propos d'un argument, n'est pas plus intéressant que de ne rien déclarer du tout, et cela est malheureusement presque toujours le cas pour nos programmes. L'idée est de transformer le programme. Par exemple, quand on sait qu'un argument a toujours l'instanciation f(G2,V4,G2,N3), on va le remplacer par trois arguments différents G2,V4,N3 pour lesquels une déclaration de mode intéressante peut maintenant être faite. Le problème est évidemment de répercuter correctement ces transformations dans tout le programme.

Cette transformation se fait en deux temps. Premièrement (a), si le $i^{ème}$ argument d'un prédicat, lors de l'appel de celui-ci, est décrit comme étant de la forme f(t1,...,tn), il faut s'assurer que cet argument a cette forme dans toutes occurrences du prédicat dans le texte du programme. Ensuite, il est alors possible de remplacer cet argument par n nouveaux arguments t1,...tn.

a. Mise en forme des prédicats.

1.Tête de clause:

Soit une clause: P(t1,...,tn) <- B1,...,Bn, un ensemble d'atomes_1, C={A1,...,Am}, qui décrivent l'instanciation du prédicat P/n au moment de l'appel et Ag=Gen(C). En se rappelant que les Gi,Vi,Ni sont des variables, on a que

-H=P(t1,...,tn) s'unifie à Ag avec mgu θ,
-H s'unifie à Aj avec mgu θj, $1 \leq j \leq m$
-H θ s'unifie à Aj avec mgu δj, $1 \leq j \leq m$
-$H\theta\delta j = H\theta j$, $1 \leq j \leq m$

Cela nous permet de remplacer H <- B1,...,Bn par $H\theta$ <- $(B1,...,Bn)\theta$ puisqu'au moins ces instanciations seront faites lors d'une exécution.

2.Corps de la clause:

Soit une clause, H <- $B1,...,B_{i-1},P(t1,...,tn),B_{i+1},...,Bn$, C={A1,...,An} un ensemble d'atomes_1 qui décrivent l'instanciation du prédicat P/n au moment de l'appel et Ag=Gen(C).

Lorsque P(t1,...,tn) sera sélectionné, son instanciation sera telle que décrite par un des Aj $\in$ C. Donc P(t1,...,tn) aura au moins l'instanciation décrite par Ag. Si θ est le mgu de P(T1,...,tn) et de Ag, on peut transformer la clause en H <- $B1,...,B_{i-1},(P(t1,...,tn),B_{i+1},...,Bn)\theta$ puisqu'au moins ces instanciations seront faites lors d'une exécution.

remarque:

Quand on unifie un atome_2 et un prédicat, on suppose que toutes les variables $\in$ A2 sont différentes des variables Prolog qui portent le même nom.

b. Aplatir les prédicats.

Supposons que dans toutes les clauses du programme, tous les prédicats pour lesquels on a de l'information sont transformés comme on vient de le voir. Alors, on est sûr que si dans l'atome_2 qui décrit de manière générale l'instanciation d'un prédicat P/n, un terme est de la forme f(I1,...,In), alors ce terme est de la forme f(t1,...,tn) dans toutes les occurrences de P/n. On va maintenant pouvoir remplacer tous les termes de la forme f(t1,...,tn) par n termes t1,...,tn.

Soit un prédicat P=P(t1,...,tn) et un atome_2 Ag=P(a1,...an), **Fg l'atome_2 Ag aplati, noté Fg=flat(Ag), et Fp le prédicat P aplati en fonction d'Ag, noté Fp=flat(P,Ag)** sont définis comme suit:

-si ∃ j, 1≤j≤n: a_j=f(a_{j1},..a_{jm}), et tj=f(t_{j1},...,t_{jm}), alors
Ag'=P(a1,...a_{j-1},a_{j1},...,a_{jm},a_{j+1},..,am),
P'=P(t1,...,t_{j-1},t_{j1},...,t_{jm},t_{j+1},...,tn) et Fg=flat(Ag'),
Fp=flat(P',Ag').
-sinon Fp=P et Fg=Ag.

Une seconde transformation consiste à enlever les termes qui correspondent à la même variable Vi.

Soit un atome_2 Ag=P(a1,...,an) et un prédicat P=P(t1,...,tn). **Ndvg, l'atome_2 Ag sans double Vi, noté Ndvg = Nd(Ag) et Ndvp, le prédicat P sans double Vi en fonction de Ag, noté Ndvp=Nd(P,Ag)** sont définis comme suit

-si ∃ ai,aj, (i<j): ai=aj=V_k alors
Fg'=P(a1,...,a_{j-1},a_{j+1},...an), Fndg=Fnd(Fg') et
Fndp=Fnd(P(t1,...,t_{j-1},t_{j+1},...,tn),Fg').
-sinon Ndvp=P et Ndvg=Ag.

Aplatir les prédicats du programme revient à remplacer tout prédicat P pour lequel on dispose d'un atome_2 Ag décrivant de manière générale son instanciation au moment de l'appel par un prédicat P'=Nd(flat(P,Ag)).

remarque:

Si dans le programme, il existe des prédicats ayant même symbole de prédicat mais d'arité différente, il se peut qu'après transformation, on obtienne deux prédicats ayant la même arité, et la correction de la transformation n'est pas assurée. Pour les prédicats créés lors de la compilation du programme, il n'y a pas de problème, car la fonction Newname(C) nous garantit qu'ils ont tous un symbole de prédicat différent. Le problème peut toutefois se poser pour les prédicats qui ont été sélectionnés pour être entièrement résolus. Pour ces prédicats, une condition nécessaire à leur transformation est qu'il n'existe pas dans le programme d'autres prédicats avec le même symbole de prédicat et d'arité différente.

c. Génération des déclarations de mode.

Après avoir réalisé ces transformations, il devient maintenant assez facile de générer des déclarations de mode. Il suffit en effet d'analyser les atomes_2 aplatis sans double, qui décrivent de manière générale l'instanciation des prédicats qui ont été transformés, et de déclarer I (input) un argument Gi ou une constante, O (ouput) un argument Vi, et ? dans les autres cas.

2.3 Conclusion.

Je viens de présenter les principes de transformation d'un programme pour générer des déclarations de mode lorsqu'on dispose d'informations sur l'instanciation des prédicats du programme lors de leur sélection par la règle de calcul. J'ai également montré comment obtenir l'information sur l'instanciation des prédicats pour des programmes construits par compilation d'informations de contrôle.

Dans le chapitre suivant je vais montrer comment utiliser la méthodologie présentée au chapitre 1 pour écrire un programme qui permet de réaliser ces transformations.

Le chapitre 4 est lui consacré à une évaluation du programme réalisé, des principes de transformation et de la compilation d'informations de contrôle.

Chapitre 3

REALISATION DU PROGRAMME

Dans ce chapitre, je vais montrer comment le programme décrit au chapitre précédent a été réalisé. Je présenterai d'abord l'architecture du programme, découpe en niveaux et en modules. Ensuite, je montrerai comment certaines procédures ont été construites. Le texte complet du programme, spécifications et algorithmes, se trouve lui en annexe.

3.1 Architecture logique.

3.1.1 Découpe en niveaux.

Le programme a été organisé selon une découpe en trois niveaux. Je vais présenter cette découpe niveau par niveau en présentant pour chaque niveau le type de procédure qu'il contient.

Niveau 0.

Au niveau le plus bas, se situent les procédures de base. On y retrouve par exemple :
-des procédures pour connaître l'instanciation d'un terme, obtenir les composants d'une structure,.. . Ces procédures sont pour la plupart des primitives prolog.
-des primitives semblables aux précédentes, mais construites pour des termes d'instanciation_0 ou des termes d'instanciation_2.
-des primitives de manipulation de listes.

Niveau 1.

Pour construire ce programme, j'ai décidé de représenter un certain nombre de données par types abstraits. Ont par exemple été représentés par types abstraits:
-l'information sur l'arbre de résolution à partir duquel le programme a été compilé,
-le programme à transformer,
-les substitutions.

Les différentes procédures d'interface de ces types abstraits, ainsi que les procédures qui calculent quelques fonctions de base, flat(T), flat(T,Tg),..., sont définies à ce niveau-ci.

Toutes les procédures définies à ce niveau permettent d'écrire le programme en manipulant directement des objets complexes sans devoir en connaître la structure interne.

Niveau 2.

Au dernier niveau on retrouve le programme proprement dit.

3.1.2 Découpe en modules.

La découpe en modules découle directement de la découpe en niveaux. Chaque niveau est découpé en un certain nombre de modules qui regroupent les procédures qui sont logiquement liées au sein de ce niveau.

Par exemple, au niveau N1 les primitives de manipulation de programmes seront regroupées en un module, les primitives de manipulation de l'arbre symbolique en un autre.

Pour chaque module, je vais indiquer le type de procédure qu'il contient. Dans certains cas j'indiquerai la représentation des données choisies, certaines définitions ou des particularités du module. La spécification de certaines procédures est également donnée.

a. Module "termes prolog,termes_0 et termes_2".

Ce module regroupe les fonctions de base qui permettent de manipuler les termes prolog, les termes_0 et les termes_2.

Etant donné la similitude de représentation entre un terme et un prédicat, les procédures définies pour un terme (resp terme_0 ou terme_2) sont applicables aux prédicats (resp atomes_0 ou atomes_2).

Certaines procédures sont spécifiques aux termes_0, ou aux termes_1. Par exemple, pour savoir si un terme_2 est une variable on doit utiliser is_Vi(T), is_Gi(T) ou is_Ni(T) et non pas var(T). S'il n'existe pas de procédure spécifique aux termes_0 ou termes_1, cela signifie que c'est la procédure définie pour un terme prolog qui est d'application, exemple compound(T).

J'ai choisi de représenter un terme_0 et un terme_2 comme suit :

Un atome_0 est représenté par un terme prolog comme suit:
-le terme_0, T0 est représenté par un atome = T0,
-l'atome_0, P(t1,...,tn) par le terme P(t1,...,tn).

Un atome_2 est représenté par un terme prolog comme suit:
-une constante est représentée par cette même constante.
-une variable Vi, i ∈ N, par la constante vi, i ∈ N,
-une variable Gi, i ∈ N, par la constante g,
-une variable Ni par la constante any,

Il découle de ce choix que :

-les atomes_2 à représenter peuvent contenir des constantes any, g, ou vi (i ∈ N).
-dans un atome_2, toutes les variables Gi et Ni sont différentes.
-Quand on applique une procédure prolog à un terme_2, ce terme_2 est toujours "ground".

b. Module "liste".

Le module "liste" regroupe quelques procédures de manipulation de listes telles que member, append, ...

Ce module offre le choix entre deux sortes de listes. La liste prolog habituelle et la liste associative.

Une liste associative est un triple (Name,Inf,Sup). Name est le nom de la liste, Inf le numéro associé au premier élément de la liste et Sup le numéro associé au dernier élément de la liste. Les éléments de la liste sont représentés par Sup-Inf+1 relation ass_1(N,Name,Elem) avec N un entier positif Inf≤N≤Sup, qui détermine la position de Elem dans la liste de nom Name.

Pour pouvoir spécifier les procédures de ce module, et plus spécialement la directionnalité, j'ai besoin d'une nouvelle notation pour décrire une liste de longueur fixe dont les éléments peuvent avoir n'importe quelle instanciation. Je ne peut dire que cette liste est "ngv", car la liste [a|X] est bien "ngv" mais pas de longueur fixe. J'introduis la notation **L(Inst)**, pour décrire une liste de longueur fixe dont les éléments ont une instanciation telle que décrite par Inst.

c. Module "programme".

Pour pouvoir générer des déclarations de mode pour un programme, j'ai besoin d'avoir accès à chaque clause de celui-ci pour pouvoir les transformer.

J'ai choisi de représenter un programme par la liste des clauses de ce programme. Cette représentation permet de pouvoir facilement écrire les procédures d'accès à ce programme. Pour permettre de changer de représentation facilement (par exemple avoir le programme dans un fichier), j'ai toutefois décidé de le repésenter par types abstraits. Par exemple la directionnalité des procédures doit tenir compte de cette possibilité de changement de représentation.

Notation: **P(Inst)** décrit un programme de longueur fixe dont les éléments ont une intanciation telle que décrite par Inst.

d. Module "descr_arbr".

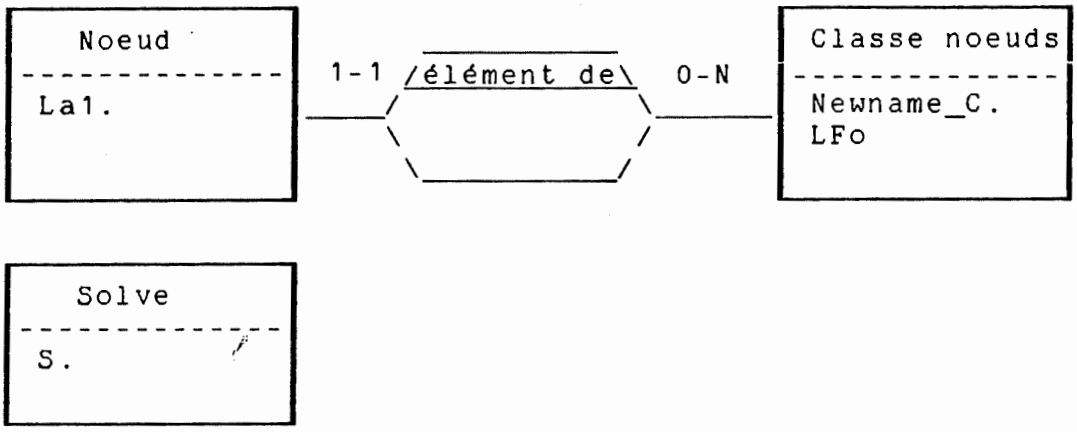
Dans la présentation du programme, j'ai essayé de séparer la partie génération de déclarations de mode de la manière dont l'information sur l'instanciation des prédicats au moment de l'appel était obtenue. Dans le cas présent cette information est obtenue à partir de l'arbre de résolution symbolique à partir duquel le programme a été compilé. Cette information pourrait être obtenue autrement [Mellish 81], [Debray 86], dans ce cas seul ce module-ci devrait être changé.

Pour pouvoir générer les déclarations de mode, j'ai besoin des informations suivantes à propos de l'arbre de résolution symbolique à partir duquel le programme a été compilé:

-l'ensemble des noeuds de l'arbre,
-pour tout noeud de l'arbre, la classe de noeuds similaires à laquelle appartient ce noeud,
-pour toute classe de noeuds similaires, les éléments de $F_{\infty}^{*}(C)$ classés par ordre.
-l'ensemble des prédicats qui ont été sélectionnés pour être entièrement résolus.

Ces informations sont appelées le descriptif de l'arbre de résolution symbolique, noté **decsr_As**.

Un descr_As peut être représenté par le schéma E_R_A suivant:



-est Noeud tout noeud de l'arbre de résolution symbolique et La1 est la liste des atomes_1 du noeud
-est Classe noeuds toute classe de noeuds similaires de l'arbre de résolution symbolique, Newname_C est le symbole de prédicat associé à la classe de noeuds et LFo est la liste des éléments de Fo*(C) classés par ordre.
-"élément de" indique l'appartenance d'un noeud à une classe de noeuds similaires.
-est Solve tout atome_1 sélectionné pour être entièrement résolu dans l'arbre de résolution symbolique, S est la représentation de cet atome_1.

L'interface du module permet d'introduire des informations sur l'arbre symbolique et de transformer le descr_arbr en une liste d'atomes_2 qui contient pour chaque classe de noeuds similaires C un atome_2 Ag=gen(C), et un atome_2 Ag=gen(ES) pour chaque ensemble ES de prédicats S ayant même symbole de prédicat et même arité.

e. **Module "substitution".**

Ce module regroupe toutes les procédures relatives à des substitutions. Je vais d'abord rappeler quelques définitions.

Une **substitution θ** est un ensemble fini de la forme {V1/t1,...,Vn/tn}, où chaque Vi est une variable, chaque ti un terme distinct de Vi et les variables V1,...,Vn sont toutes distinctes. Chaque élément Vi/ti est appelé un lien pour Vi.[Lloyd 87].

Soit θ={u1/t1,...,um/sm} et σ={v1/t1,...,vn/tn} des substitutions. Alors **la composition θσ de θ et σ** est la substitution obtenue de l'ensemble {u1/t1σ,...,um/smσ,v1/t1,...,vn/tn} en supprimant les liens ui/tiσ pour lesquels ui=tiσ et en supprimant les liens vj/tj pour lesquels vj ∈ {u1,...,um}.[Lloyd 87].

Soit un terme t, **var(t)** est l'ensemble des variables qui font partie du terme t.
Soit θ={v1/t1,...,vn/tn} une substitution, et V un ensemble de variables. **θ↓V , la substitution θ relative à l'ensemble V**, est l'ensemble des liens vi/ti ∈ θ tel que vi ∈ V.

J'ai choisi de représenter une substitution par une liste de liens (V/Term). Pour faciliter la procédure qui calcule la composition de deux substitutions, j'ai décidé de ne pas forcer les contraintes "ti est différent de Vi" et "V1,...,Vn sont toutes des variables distinctes". Les procédures qui forment l'interface du module permettent de rendre ce choix transparent.

Notation: **S(Inst)** décrit une substitution de longueur fixe dont les éléments ont une instanciation telle que décrite par Inst.

Ce module comprend entre autre les procédures unifier/3 et unifier_2/3. Comme ces deux procédures me serviront d'exemples dans la suite du chapitre en voici la spécification.

procédure unifier(T1,T2,M).

Soit
T1,T2 deux termes,
M une substitution.

Cette procédure détermine si

T1 et T2 s'unifient avec mgu M.

Directionnalité.

In(any,any,var) Out(any,any,S(ngv)). <0,1>.

****

procédure unifier_2(T,Tg,Subst).

Soit
T un terme,
Tg un terme_2,
Subst une substitution.

Cette procédure détermine si

T s'unifie à Tg avec mgu θ. Subst=θ:var(T).

Directionnalité.

In(any,gr,var) Out(any,gr,S(ngv)). <0,1>.

f. Module "fonction de base".

Ce module contient une procédure qui permet de calculer quelques fonctions pour transformer un prédicat comme flat(A2) et flat(P,Ag) ou Nd(A2) et Nd(P,A2). Ces fonctions ont été regroupées en une seule procédure pour éviter d'examiner plusieurs fois le même terme. Avnt d'en donner la spécification je vais d'abord donner une définition qui permet de simplifier celle-ci:

Soit Ag un atome_2, **M est une déclaration de mode pour l'ensemble des prédicats dont l'instanciation est décrite par Ag, noté M=de_mode(Ag)** si M est l'atome_2 A2=ndv(flat(Ag)) où chaque occurrence d'une variable Vj est remplacée par o, Gj ou une constante par i, Nj par ?.

procédure fl_ndv(P,Subst,P1,Fp,Mode).

Soit
P, Fp, Mode des prédicats,
P1 un atome_2,
Subst une substitution,

Cette procédure détermine si

Fp=ndv(flat(P Subst,P1),flat(P1)).

Mode = de_mode(P1).

Directionnalité.

In(novar,S(novar),gr,var,var)
Out(novar,S(novar),gr,novar,gr). <0,1>.

g. Module "transformation programme".

C'est dans ce module que se trouvent les procédures qui permettent de transformer le programme pour générer des déclarations de mode et de générer ces déclarations de mode.

Pour pouvoir facilement spécifier les procédures du module je vais d'abord définir quelques fonctions :

Soit P un prédicat, Sg un ensemble d'atomes_2, Fp **le prédicat P transformé en fonction de Sg, noté Fp=tr_pred(P,Sg)**, est défini comme suit:
-s'il existe Ag ∈ Sg tel que Ag a même symbole de prédicat que P alors si P et Ag s'unifient avec mgu θ, P'=θ:var(P),
Fp=ndv(flat(P',Ag),flat(Ag)),
-s'il n'existe pas Ag ∈ Sg tel que Ag a même symbole de prédicat que P alors Fp=P.

Soit C une conjonction de prédicats, Sg un ensemble d'atomes_2, **Fc la conjonction C transformée en fonction de Sg, noté Fc=trans_conj(C,Sg)** est définie comme suit:
-si C=P alors Fc=tr_pred(P,Sg),
-si C=P1,P2,...,Pn et qu'il existe Ag ∈ Sg tel que Ag a même symbole de prédicat que P alors si P et Ag s'unifient avec mgu θ, θ={(V1/T1),...,(Vm,Tm)}, Fp1=tr_pred(P1,{Ag}) et Fc=V1=T1,...,Vm=Tm,Fp1,trans_conj((P2,...,Pn)θ,Sg),
-si C=P1,P2,...,Pn et qu'il n'existe pas de Ag ∈ Sg tel que Ag a même symbole de prédicat que P,
Fc=P1,trans_conj((P2,...,Pn)).

Soit Cl une clause, Sg un ensemble d'atomes_2, **Fcl la clause Cl transformée en fonction de Sg, noté Fcl=trans_cl(Cl,Sg)**, est définie comme suit:
-si Cl= H:-T alors Fcl=tr_pred(H,Sg) :- tr_conj(T,Sg),
-si Cl=H alors Fcl=tr_pred(H,Sg).

Finalement je vais donner la spécification du programme:

procédure gen_mode(P,As,Pnew,Mode).

Soit
P,Pnew des programmes,
As un descr_arb,
Mode une liste de déclarations de mode.

Cette procédure détermine si

Sg est une liste qui possède un atome_2 qui décrit de manière générale l'instanciation de tous les prédicats dont l'instanciation au moment de l'appel est décrite dans AS.

Pnew est le programme P ou chaque clause est transformée en fonction de Sg. Mode est la liste des déclarations de mode pour Pnew.

Directionnalité.

In(P(ngv),gr,var,var) Out(P(ngv),gr,P(ngv),gr). <1,1>.

****

3.2 Construction des procédures.

3.2.1 Algorithme logique.

Après avoir présenté l'architecture du programme, je vais montrer comment certaines procédures ont été construites.

Je me limiterai à la construction de trois procédures :

-unifier_2/3 est assez spéciale étant donné la représentation des variables_2.
-Unifier/3 est présentée pour permettre de comprendre plus facilement la construction de unifer_2/3.
-fl_ndv/5 est la procédure centrale de transformation du programme.

a. Procédure unifier(T1,T2,M).

Dans le programme, je ne peut utiliser la procédure d'unification de prolog. En effet quand je transforme la clause : H <- T1,...,T_{i-1},T_i,..Tn. en
H <- T1,...T_{i-1},(Ti,...,Tn)θ., je désire que H,T1,...,T_{i-1} ne soient pas modifiés. Or, si Ti a une variable en commun avec H <- T1,...,T_{i-1} et que j'utilise l'unification de prolog, cette variable est modifiée dans toute la clause.

Cette procédure peut être écrite sans induction, en analysant les différents cas possibles. Pour unifier deux structures ayant même foncteur et même arité, je me sers d'une procédure pour unifier la liste des termes de ces deux structures.

```
unifier(T1,T2,M) <=>
          var(T1) & var(T2) & T1==T2 & empty_sub(M),

       V  var(T1) & var(T2) & ¬T1==T2 & L_to_sub([(T1/T2)],M)

       V  var(T1) & nonvar(T2) & L_to_sub([(T1/T2)],M)

       V  nonvar(T1) & var(T2) & L_to_sub([(T2/T1)],M)

       V  atomic(T1) & atomic(T2) & T1=T2 & empty_sub(M)

       V  compound(T1) & compound(T2)&
          T1=..[F|L1] & T2=..[F|L2] & unifier_L(L1,L2,M).
```

où L_to_sub(L,Sub) permet de transformer la liste de liens L en une substitution Sub.

Voyons maintenant la spécification de unifier_L/3 et comment cette procédure est construite.

procédure unifier_L(L1,L2,M)

Soit
L1,L2 deux listes,
M une substitution.

Cette procédure détermine si

M est le mgu de L1 et L2.

Directionnalité.

```
In(list(any),list(any),var)  Out(list(any),list(any),sub(ngv)).
                                                    <0,1>.
```

****

En choisissant L1 comme paramètre d'induction, "est un suffixe propre" comme relation bien fondée et L1=[], L1=[H1|T1] les deux formes structurelles de L1, on obtient facilement l'algorithme suivant:

```
unifier_L(L1,L2,M) <=>
          L1=[] & L2=[] & empty_sub(M).

      V   L1=[H1|T1] & L2=[H2|T2] &
          unifier(H1,H2,Mh) &
          substitue(T1,Mh,N1) & substitue(T2,Mh,N2)&
          unifier_L(N1,N2,Mt) & comp(Mh,Mt,M).
```

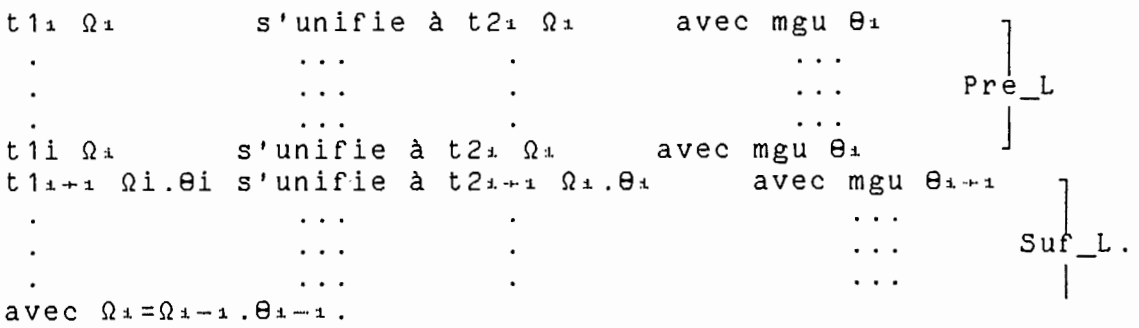
Cet algorithme a toutefois deux défauts majeurs :

-on calcule N1 (T1 Mh) et N2 (T2 Mh) à chaque récursion. En généralisant la procédure, il est possible de ne calculer que (H1 S') et (H2 S') à chaque récursion ce qui représente un gain assez important.
-quand on le transforme pour obtenir une procédure prolog on remarque qu'il est nécessaire que la procédure comp/3 soit

correcte pour la directionnalité :
In(S(novar),var,var) Out(S(novar),S(novar),S(novar)) pour obtenir la récursion terminale.

Voyons comment cette procédure peut être généralisée pour résoudre ces problèmes.

Un état de calcul peut être décrit comme suit :



Si Suf_L est vide alors M est θi. Sinon on essaie de passer à un état de calcul où Suf_L est plus petit selon notre relation bien fondée. On remarque que la seule information nécessaire à propos de Pre_L est Ωi et que changer d'état de calcul revient à calculer θi+1.

On obtient la procédure suivante:

procédure unifier_L(L1,L2,Mint,M).

Soit
L1,L2 deux listes,
Mint,M deux substitutions.

Cette procédure détermine si

L1 Mint s'unifie à L2 Mint avec mgu M.

Directionnalité.

In(L(any),L(any),S(ngv),Var) Out(L(any),L(any),S(ngv),S(ngv)).

On remarque que si Mint est la substitution vide unifier/4 est bien une généralisation de unifier/3.

```
unifier_L(L1,L2,M) <=>
          unifier_L(L1,L2,Mint,M) & empty_sub(Mint).
```

L'algorithme peut être simplement construit avec une induction sur L1 ou L2.

```
unifier_L(L1,L2,Mint,M) <=>
          L1=[] & L2=[] & M=Mint.

      V   L1=[H1¦T1] & L2=[H2¦T2] &
          substitue(H1,Mint,N1) & substitue(H2,Mint,N2) &
          unifier(N1,N2,Mh) & comp(Mint,Mh,Mint1) &
          unifier_l(T1,T2,Mint1,M).
```

remarque:

-comme annoncé plus haut on ne calcule que (H1 Mint) et (H2 Mint) à chaque récursion.
-on a uniquement besoin d'une version correcte de comp/3 pour la directionnalité
In(S(ngv),S(ngv),var) Out(S(ngv),S(ngv),S(ngv)) pour obtenir la récursion terminale lors de la transformation de l'algorithme logique en une procédure prolog.

b. Procédure unifier_2(T,Tg,Subst).

Dans le programme on a besoin d'une deuxième procédure d'unification.

Cette procédure ressemble à unifier(T,T1,Mgu), mais en diffère principalement par les deux caractéristiques suivantes :

-On ne s'intéresse pas à θ, le mgu de T et Tg mais uniquement à $\theta_{|var(T)}$ la substitution θ relative à l'ensemble des variables de T.
-les variables_2 sont représentées par des termes de base, or pour l'unification je dois les considérer comme des variables prolog différentes de toutes les variables prolog ayant le même nom. Pour ce faire, je remplace chaque variable_2 par une variable prolog avant de chercher $\theta_{|var(T)}$.

Comme pour unifier/3, je vais me servir d'une procédure unifier_L2/4 quand T ou Tg est une structure. Je vais cependant directement donner la spécification et construire la version généralisée.

procédure unifier_L2(L,Lg,Sint,Subst,Lvt,Lvi).

Soit
L une liste,
Lg une liste de termes_2,
Sint,Subst deux substitutions,
Lvt,Lvi.

pre: Lvt et Lvi sont des listes de termes (V/Term) où V est une variable_2 Vi. Si dans une liste il existe deux termes (V1/T1) et (V2/T1) alors V1 est différent de V2.

Cette procédure détermine si

Evi(Lg) est l'ensemble des variables Vi qui apparaissent dans Lg.

Lg' est la liste Lg où chaque variable Vi ∈ Evi(Lg) est remplacée par le terme t si ∃ un couple (Vi,t) ∈ Lvt.

Soit θ le mgu de (L Sint) et de (Lg' Sint), Sl=$\theta_{|var(L)}$ et Subst=Sint.Sl

Lv' est une liste qui contient un terme (Vj/tj) pour chaque variable Vj qui apparaît dans Evi(Lg) et pas dans un terme

(v/t) de Lvt. Si ∃ un lien (Vj/term) dans θ:Evi(L,Lg) alors Tj est tel que Tj.Subst= term, sinon Tj est une variable qui n'est ni élément de var(L.Sint) ni élément de var(Lvt).

Lvi est la concaténation de Lvt et Lv'.

****

Sint et Lvt sont les informations à propos de ce qu'on a déjà fait quand on généralise unifier_L2/3. Lvt est une liste un peu hybride qui permet de toujours remplacer une variable_2 par la même variable prolog. Svt sert aussi à tenir compte des substitutions relatives aux variables de Tg.

En choisissant L comme paramètre d'induction on obtient la procédure :

```
unifier_L2(L,Lg,Sint,Subst,Lvt,Lvi) <=>
          L=[] & Lg=[] & Subst=Sint & Lvi=Lvt

          L=[H|T] & Lg=[Hg|Tg] &
          unifier_2(H,Hg,Sint,Sint1,Lvt,Lvt1) &
          unifier_L2(T,Tg,Sint1,Subst,Lvt1,Lvi).
```

Pour finir, il nous reste encore à spécifier et à construire unifier_2/6. Cette procédure est une généralisation de unifer_2/3 équivalente à la généralisation de unifier_L2/6 par rapport à unifier_L2/3.

procédure unifier_2(T,Tg,Sint,Subst,Lvt,Lvi).

Soit
- T un terme,
- Tg un terme_2,
- Sint,Subst deux substitutions,
- Lvt,Lvi.

pre: Lvt et Lvi sont des listes de termes (V/Term) où V est une variable_2 Vi. Si dans une liste il existe deux couples (V1/T1) et (V2/T2) alors V1 est différent de V2.

Cette procédure détermine si

Evi(Tg) est l'ensemble des variables Vi qui apparaissent dans Tg.

Tg' est Tg où chaque variable Vi est remplacée par le terme t si ∃ un couple (Vi,t) ∈ Lvt.

Soit θ le mgu de (T Sint) et de (Tg' Sint), St=θ:var(T) et Subst=Sint.St

Lv' est une liste qui contient un terme (Vj/Tj) pour chaque variable Vj qui apparaît dans Evi(Tg) et pas dans un terme

(v/t) de Lvt. Si ∃ un lien (Vj/term) dans θ=mgu(Tg) alors Tj est tel que Tj.Subst= term, sinon Tj est une variable qui n'est ni élément de var(T.Sint) ni élément de var(Lvt).

Lvi est la concaténation de Lvt et Lv'.

****

Cette procédure est une généralisation de unifier_2/4 :

```
unifier_2(T,Tg,Subst) <=>
          unifier_2(T,Tg,[],Subst,[],V).
```

Elle peut être construite sans induction.

```
unifier_2(T,Tg,Sint,Subst,Lvt,Lvi) <=>
          is_vi(Tg) &  member((Tg/Term),Lvt) &
          substitue(T,Sint,T1) & substitue(Term,Sint,Term1) &
          unifier(T,Term1,Mt) &
          comp(Sint,Mt,Subst) & Lvi=Lvt.

      V   is_vi(Tg) &
          ¬ ∃  Term, Sv1 : member((Tg/Term),Lvt,Sv1) &
          member((Tg/T),Lvi,Lvt) & Subst=Sint,

      V   (is_gi(Tg) V is_ni(Tg)) & Subst=Sint & Lvi=Lvint,

      V   is_const(Tg) & substitue(T,Sint,T1) &
          unifier(T1,Tg,Mt) & comp(Sint,Mt,Subst) &
          Lvt=Lvi

      V   compound(Tg) & substitue(T,Sint,T1) & var(T1) &
          functor(Tg,F,N) & functor(X,F,N) &
          unifier_2(X,Tg,[],Sx,Lvt,Lvi) &
          substitue(X,Sx,X1) & L_to_sub([T1/X1],St),
          comp(Sint,St,Subst)

       V  compound(Tg) & substitue(T,Sint,T1) & compound(T1)&
          Tg=..[F|Lg) & T=..[F|L] &
          unifier_L2(Lg,L,Sint,Subst,Lvt,Lvi).
```

c. Procédure fl_ndv(T,Subst,T1,Ft,Mode).

Pour construire fl_ndv/5 (spécifiée dans le module "fonction de base"), je me sers d'une procédure pour obtenir la liste des arguments de Ft à partir de la liste des arguments de T et de la liste des arguments de T1. Je vais directement donner une version généralisée de cette procédure. Avant d'en donner la spécification, je vais d'abord définir deux fonctions.

Soit L=[t1,...,tn] une liste de termes, L1=[a1,...,an] une liste de termes_2, les fonctions **flat_Lt(L,L1)=Fl et flat_Lt(L1)**=Fl1 sont définies comme suit :
-si ∃ j, 1≤j≤n: $a_j=f(a_{j1},..a_{jm})$, et $t_j=f(t_{j1},...,t_{jm})$, alors
 L1'=[a1,...a_{j-1},a_{j1},...,a_{jm},a_{j+1},..,am],
 L'=[t1,...,t_{j-1},t_{j1},...,t_{jm},t_{j+1},...,tn] et Fl1=flat_Lt(L1'),
 Fl=flat_Lt(L',L1').
-sinon Fl=L et Fl1=L1.

Soit L=[t1,...,tn] une liste de termes, L1=[a1,...,an] une liste de termes_2 et Lvi une liste de variables_2 Vi, les fonctions **ndv_Lt(L,L1,Lvi)=N et ndv_Lt(L1,Lvi)=N1** sont définies comme suit :

-si ∃ ai,aj (i<j): ai=aj=Vk alors L1'=[a1,...,aj-1,aj+1,...an], N1=Ndv_Lt(L1',Lvi) et N=ndv_Lt([t1,...,tj-1,tj+1,...,tn],L1',Lvi).

-si ∃ ai: ai ∈ Lvi alors L1'=[a1,...,aj-1,aj+1,...an], N1=Ndv_Lt(L1',Lvi) et N=ndv_Lt([t1,...,tj-1,tj+1,...,tn],L1',Lvi).

-sinon N=L et N1=L1.

procédure f_ndv_L(L,Subst,L1,Fl,Lmode,Lvint,Lvi).

Soit
L,Fl,Lmode des listes,
L1 une liste de termes_2,
Subst une substitution.

Cette procédure détermine si

Fl=ndv_Lt(flat_Lt((L Subst),L1), flat_Lt(L1), Lvint).

Lmode est la liste ndv_Lt(flat_Lt(L1),Lvi) où toute occurrence d'une variable Vj est remplacée par o, Gj ou une constante par i, Nj par ?.

Lvi est la liste des variables_2 Vi qui apparaissent dans ndv_Lt(flat_Lt(L1),Lvi) concaténée à Lvint.

****

La procédure est construite par induction sur L.

```
f_ndv_L(L,Subst,L1,Fl,Lmode,Lvint,Lvi) <=>
     L=[] & L1=[] & FL=[] & Lmode=[] & Lvi=Lvint.

  V  L=[H0|T] & L1=[H1|T1] &
     substitue(H0,Subst,H) &
     compound(H) & compound(H1) &
     H=..[F|Lh] & H1=..[F|Lh1] & empty_sub(S) &
     f_ndv_L(Lh,S,Lh1,Fh,Hmode,Lvint,Lvint1) &
     f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint1,Lvi) &
     append(Fh,Ft,Fl) & append(Hmode,Tmode,Lmode).

  V  L=[H|T] & L1=[H1|T1] &
     is_vi(H1) & member(H1,Lvint) &
     f_ndv_L(T,Subst,T1,Fl,Fmode,Lvint,Lvi)

  V  L=[H0|T] & L1=[H1|T1] &
     substitue(H0,Subst,H) &
     is_vi(H1) & ¬ member(H1,Lvint) &
     Fl=[H|Ft] & Lmode=[o|Tmode] & Lvint1=[H1|Lvint] &
     f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint1,Lvi)

  V  L=[H0|T] & L1=[H1|T1] &
     substitue(H0,Subst,H) &
     (is_Gi(H1) V is_const(H1)) &
     Fl=[H|Ft] & Lmode=['I'|Tmode] &
     f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi)
```

```
V  L=[H0|T] & L1=[H1|T1] &
   substitue(H0,Subst,H) &
   is_Ni(H1)
   Fl=[H|Ft] & Lmode=[?|Tmode] &
   f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi)
```

La procédure Fl_ndv/5 peut maintenant être facilement écrite.

```
fl_ndv(P,Subst,P1,Fp,Mode) <=>
     P=..[F|L] & P1=..[F|L1] &
     f_ndv_L(L,Subst,L1,Fl,Lmode,[],Lvi) &
     Fp=..[F|Fl] & Mode=..[F|Lmode].
```

3.2.2 Procédures logiques.

Je vais maintenant me servir de la procédure fl_ndv_L/7 pour montrer les différentes étapes suivies pour obtenir une procédure prolog à partir d'un algorithme logique. Je vais d'abord donner la procédure logique obtenue en numérotant les clauses. Ces clauses me serviront d'exemples pour expliquer certaines étapes.

Cette partie est assez courte. J'estime en effet que la partie créative de la construction du programme se situe au niveau de la spécification des procédures et de la construction des algorithmes logiques.

a. Algorithme obtenu.

Je désire obtenir une version correcte de
fl_ndv_L(L,Subst,L1,Fl,Lmode,Lvint,Lvi). pour la
directionnalité: In(L(ngv),S(ngv),gr,var,var,L(ngv),var)
Out(L(ngv),S(ngv),gr,L(ngv),gr,L(ngv),L(ngv)) <0,1>.

Pour expliquer les transformations effectuées pour obtenir la procédure prolog à partir de l'algorithme logique, je vais donner la procédure finalement obtenue et expliquer comment certaines clauses ont été dérivées.

```
(1)f_ndv_L([H|T],Subst,[H1|T1],Fl,Lmode,Lvint,Lvi) :-
          is_vi(H1),
          member(H1,Lvint),
          !,
          f_ndv_L(T,Subst,T1,Fl,Fmode,Lvint,Lvi).
(2)f_ndv_L([H0|T],Subst,[H1|T1],Fl,Lmode,Lvint,Lvi) :-
          is_vi(H1),
          !,
          substitue(H0,Subst,H),
          Fl=[H|Ft],
          Lmode=[o|Tmode],
          Lvint1=[H1|Lvint],
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint1,Lvi).
```

```
(3)f_ndv_L([H0|T],Subst,[any|T1],Fl,Lmode,Lvint,Lvi) :-
          !,
          substitue(H0,Subst,H),
          Fl=[H|Ft],
          Lmode=[?|Tmode],
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi).
(4)f_ndv_L([H0|T],Subst,[H1|T1],Fl,Lmode,Lvint,Lvi) :-
          atomic(H1),
          !,
          substitue(H0,Subst,H),
          Fl=[H|Ft],
          Lmode=[i|Tmode],
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi).
(5)f_ndv_L([H0|T],Subst,[H1|T1],Fl,Lmode,Lvint,Lvi) :-
          substitue(H0,Subst,H),
          H=..[F|Lh] & H1=..[F|Lh1],
          f_ndv_L(Lh,[],Lh1,Fh,Hmode,Lvint,Lvint1),
          append(Fh,Ft,Fl),
          append(Hmode,Tmode,Lmode),
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint1,Lvi).
(6)f_ndv_L([],_,[],[],[],Lvi,Lvi).
```

b. Démarche.

1.L'algorithme logique est d'abord transformé en un ensemble de clauses en appliquant les transformations initiales. Toutes les transformations possibles ne sont pas nécessairement faites. Par exemple j'ai gardé (is_gi(H1) V is_const(H1)) dans la 4ième clause.

2.Certains litéraux sont dépliés à l'aide de la transformation basée sur l'évaluation partielle et la transformation basée sur l'égalité.
exemple:

- -dans la clause 3, is_any(H1) est remplacé par H1=any.
- -dans la clause 4, is const(H1) est remplacé par not(H1=g), not(H1=any), not(is_vi(H1)), atomic(H1). De cette manière j'obtiens (not(H1=any), not(is_vi(H1)), atomic(H1)) à partir de(is_g(H1) V is_const(H1)).

3.Une permutation qui permet d'obtenir la récursion terminale est choisie. J'essaie également d'avoir les égalités X=t (X une variable qui apparaît dans la tête) le plus près de la tête de la clause. Les prédicats qui peuvent échouer sont placés le plus tôt possible. Je regarde si cette permutation est correcte par rapport à la directionnalité et au type.

exemple:

pour la clause 3, j'obtiens:

```
f_ndv_L(L,Subst,L1,Fl,Lmode,Lvint,Lvi) :-
          L=[H0|T],
          L1=[H1|T1]
          H1=any
          substitue(H0,Subst,H),
          Fl=[H|Ft],
          Lmode=[?|Tmode],
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi).
```

qui est correcte si on dispose d'une version correcte de substitue/3 pour la directionnalité
In(ngv,S(ngv),var) Out(ngv,S(ngv),any)..

4.Certains "not" sont supprimés grâce aux transformations basées sur des arbres de résolution équivalents. Cela entraîne souvent l'obligation de réordonner les clauses.

exemple:
-dans la deuxième clause not(member(H1,Lint)) peut être supprimé en plaçant un cut dans la première clause.
-dans la clause 4 not(is_vi(H1)) et not(H1=any) peuvent être supprimés en plaçant un cut dans la deuxième et la troisième clause.

5.Je vérifie que l'ordre des clauses est correct par rapport au critère de terminaison. Dans le programme ce critère n'a jamais posé de problème car les procédures ont toujours été écrites avec le paramètre d'induction "ground", et dans aucun cas je n'ai eu à traiter des arbres de résolution infinis.

6.Les égalités sont ramenées dans la tête quand cela est possible grâce aux transformations basées sur l'égalité.

exemple: la clause 3 est obtenue à l'aide de cette transformation à partir de :

```
f_ndv_L(L,Subst,L1,Fl,Lmode,Lvint,Lvi) :-
          L=[H0|T],
          L1=[H1|T1]
          H1=any
          substitue(H0,Subst,H),
          Fl=[H|Ft],
          Lmode=[?|Tmode],
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi).
```

3.3 Exemples.

Pour terminer le chapitre, je vais donner deux exemples de programmes qui ont été transformés à l'aide de l'application réalisée. Dans ces exemples, je présente le programme de départ, puis le programme après compilation des informations de contrôle et enfin les déclarations de mode et le programme obtenus après transformations.

Exemple 1 : Tri.

Si on définit une nouvelle règle de calcul pour le programme de tri par permutation :

```
sort(X,Y) :-
     perm(X,Y),
     ord(Y).

perm([],[]).
perm([X|Y],[U,V]) :-
     del(U,[X|Y],W),
     perm(W,V).

del(X,[X|Y],Y).
     del(X,[Y|Z],[Y|W]) :-
      del(X,Z,W).

ord([]).
ord([X]).
ord([X,Y|Z]) :-
     X ≤ Y,
     ord([Y|Z]).
```

et si on compile cette règle de calcul on obtient le programme suivant :

```
sort_new( [], []).
sort_new( [_x|_y], [_u|_v]) :-
    del( _u, [_x|_y], _w),
    p( perm( _w, _v), ord( [_u|_v])).

p( perm( [], []), ord( [_u])).
p( perm( [_x|_y],[_u2|_v]), ord([_u1,_u2|_v])) :-
    del(_u2,[_x|_y], _w),
    _u1 =< _u2,
    p( perm( _w, _v), ord([_u2|_v])).

del(_x,[_x|_y],_y).
del(_x,[_y|_z],[_y|_w]) :-
                del(_x,_z,_w).
```

A partir de l'arbre de résolution symbolique, on obtient les atomes_2 qui décrivent l'instanciation des prédicats du programme lors de leur sélection par la règle de calcul :

```
sort_new(g,v1),
del(v1,g,v2),
p(perm(g,v3), ord([g|v3]))
```

il est alors possible de transformer ce programme pour obtenir les déclarations de mode et le programme suivant :

```
 del(o,i,o).
 p(i,o,i).
 sort_new(i,o).

 sort_new([],[]).
 sort_new([_x|_y],[_u|_v])
       del(_u,[_x|_y],_w)
       p(_w,_v,_u).

 p([],[],_u).
 p([_x|_y],[_u2|_v],_u1)
       del(_u2,[_x|_y],_w)
       _u1=<_u2
       p(_w,_v,_u2).

del(_x,[_x|_y],_y).
del(_x,[_y|_z],[_y|_w]) :-
                del(_x,_z,_w).
```

Exemple 2 : Lucky number.

Si on définit une nouvelle règle de calcul pour le programme lucky(N,L) qui pour un entier N permet d'obtenir les N entiers déterminés par l'algorithme du crible d'Ulman:

```
lucky(N,L) :-
     odd(3,I),
     sift(I,3,L),
     len(L,N).

odd(N,[]).
odd(N,[N|I]) :-
     N1 is N+2,
     odd(N1,I).

sift([],P,[]).
sift([L|I],P,[L|S]) :-
     fil(I,L,P,R),
     P1 is P+1,
     sift(R,P1,S).

fil([],L,P,[]).
fil([N|I],L,L,R) :-
     fil(I,L,1,R).
fil([N|I],L,P,[N|R]) :-
     L \= P,
     P1 is P+1,
     fil(I,L,P1,R).

len([],0).
len([H|T],N) :-
     n>0,
     N1 is N-1,
     len(T,N1).
```

et si on compile cette règle de calcul on obtient le programme suivant :

```
lucky_new( 0, []) .
lucky_new( _x, [3¦_y]) :-
        _z is 3 + 2 ,
        _t is 3 + 1 ,
        _u is _x - 1 ,
        _x > 0 ,
        p_0(len(_y,_u),
odd(_z,_v),sift(_w,_t,_y),[fil(_v,3,3,_w)]) .

p_0( _x, odd( _y, []), _z, _t) :-
        append( _u, [_v], _t) ,
        p_F2( _u, _v, _x, _z) .
p_0( _x, odd( _y, [_y¦_z]), _t, _u) :-
        spips( _u, fil( _v, _, _, _), _v, [_w], _p),
        _q is _y + 2 ,
        p_F( _w, _x, odd( _q, _z), _t, _p) .

p_F( fil( [_x¦_y], _z, _z, _t), _u, _v, _w, _p) :-
        p_0( _u, _v, _w, [fil( _y, _z, 1, _t)¦_p]) .
p_F( fil( [_x¦_y], _z, _t, [_x¦_u]), _v, _w, _p, _q) :-
        spips( _q, fil( _r, _, _, _), _r, [_s], _m),
        _n is _t + 1 ,
        _z \= _t ,
        p_F( _s, _v, _w, _p, [fil( _y, _z, _n,_u)¦_m]) .
p_F( fil( [_x¦_y], _z, _t, [_x¦_u]), _v, _w, _p, _q) :-
        spips( [_p], sift( _r, _, _), _r, [_s], []),
        _m is _t + 1 ,
        _z \= _t ,
        p_S_L( _v, _w, _s, [fil( _y, _z, _m, _u)¦_q]) .

p_S_L( len([_x¦_y], _z), _t, sift( [_u¦_v], _w,[_u¦_p]), _q) :-
      _r is _w + 1 ,
      _s is _z - 1 ,
      _z > 0 ,
      p_0(len(_y, _s), _t, sift(_m, _r,
_p),[fil(_v,_u,_w,_m)¦_q]) .

p_F2( _x, fil( [], _y, _z, []), _t, _u) :-
        append( _v, [_w], _x) ,
        p_F2( _v, _w, _t, _u) .
p_F2( [], fil( [], _x, _y ,[]), len( [], 0),sift([],_g, [])).

spips([] , _ , _ , [] , [] ) .
spips( [ _shape ¦ _Tl] , _shape , _parm , [ _shape] , _Tl):-
            nonvar( _parm) ,
            ! .
spips([_Hd¦ _Tl], _shape, _parm, _l1,[ _Hd ¦ _Tl1]):-
            spips( _Tl , _shape , _parm , _l1 , _Tl1 ) .

append([], _l,_l) .
append([_h ¦_t], _l, [_h ¦ _t1] ):-
            append(_t, _l, _t1) .
```

A partir de l'arbre de résolution symbolique, on obtient les atomes_2 qui décrivent l'instanciation des prédicats du programme lors de leur sélection par la règle de calcul :

```
p_0(any,odd(g,v2),any,any) .
p_F(fil([g|v19],g,g,v14),any,any,any,any) .
p_S_L(len(v5,g),any,sift([g|v2],g,v5),any) .
p_F2(any,fil(any,g,g,any),any,any))
```

il est alors possible de transformer ce programme pour obtenir les déclarations de mode et le programme suivant :

```
  p_F2(?,?,i,i,?,?,?).
  p_S_L(o,i,?,i,o,i,?).
  p_F(i,o,i,i,o,?,?,?,?).
  p_0(?,i,o,?,?).

lucky_new(0,[]).
lucky_new(_x,[3|_y]) :-
      _Z is 3+2,
      _t is 3+1,
      _u is _x-1,
      _x>0,
      p_0(len(_y,_u),_z,_v,sift(_w,_t,_y),[fil(_v,3,3,_w)]).

p_0(_x,_y,[],_z,_t) :-
      append(_u,[_v],_t),
      _v=fil(_v1,_v2,_v3,_v4),
      p_F2(_u,_v1,_v2,_v3,_v4,_x,_z).
p_0(_x,_y,[_y|_z],_t,_u) :-
      spips(_u,fil(_v,_a1,_a2,_a3),_v,[_w],_p),
      _q is _y+2,
      _w=fil([_w1|_w2],_w3,_w4,_w5),
      p_F(_w1,_w2,_w3,_w4,_w5,_x,odd(_q,_z),_t,_p).

p_F(_x,_y,_z,_z,_t,_u,_v,_w,_p) :-
      _v=odd(_v1,_v2),
      p_0(_v,_v1,_v2,_w,[fil(_y,_z,1,_u)|_m]).
p_F(_x,_y,_z,_t,[_x|_u],_v,_w,_p,_q) :-
      spips(_q,fil(_r,_a1,_a2,_a3),_r,[_s],_m),
      _n is _t+1,
      _z\=_t,
      _s=fil([_s1|_s2],_s3,_s4,_s5),
      p_F(_s1,_s2,_s3,_s4,_s5,_v,_w,_p,[fil(_y,_z,_n,_u)|_m]).
p_F(_x,_y,_z,_t,[_x|_u],_v,_w,_p,_q) :-
      spips([_q],sift(_r,_a1,_a2),_r,[_s],[]),
      _m is _t+1,
      _z\=_t,
      _v=len(_v1,_v2),
      _s=sift([_s1|_s2],_s3,_s4),
      p_S_L(_v1,_v2,_w,_s1,_s2,_s3,[fil(_y,_z,_m,_u)|_q]).

p_S_L([_x|_y],_z,_t,_u,_v,_w,_q) :-
      _r is _w+1,
      _s is __z-1,
      _z>0,
      _t=odd(_t1,_t2),
      p_0(len(_y,_s),_t1,_t2,sift(_m,_r,_p2),
                                    [fil(_v,_u,_w,_m)|_q]).
```

```
p_F2(_x,[],_y,_z,[],_t,_u) :-
       append(_v,[_w],_x),
       _w=fil(_w1,_w2,_w3,_w4),
       p_F2(_v,_w1,_w2,_w3,_w4,_t,_u),
p_F2([],[],_x,_y,[],len([],0),sift([],_g,[])).

spips([] , _ , _ , [] , [] ) .
spips( [ _shape | _Tl] , _shape , _parm , [ _shape] , _Tl):-
              nonvar( _parm) ,
              ! .
spips([_Hd| _Tl], _shape, _parm, _l1,[ _Hd | _Tl1]):-
              spips( _Tl , _shape , _parm , _l1 , _Tl1   .

append([], _l,_l) .
append([_h |_t], _l, [_h | _t1] ):-
              append(_t, _l, _t1) .
```

Chapitre 4

Evaluation du programme

Au chapitre 2, j'ai présenté une méthode qui permet de compiler des informations de contrôle. J'ai également vu comment il était possible de se servir de l'information utilisée lors de la "compilation" du programme pour obtenir les différentes instanciations des prédicats du nouveau programme au moment de leur sélection par la règle de calcul. Finalement, j'ai montré comment il était possible de générer des déclarations de mode à partir des différentes instanciations des prédicats au moment de l'appel et comment transformer le programme quand il n'est pas possible de générer des déclarations de mode intéressantes à partir de l'information dont on dispose.

Après avoir fait quelques remarques complémentaires sur la compilation d'informations de contrôle, je vais montrer dans quels cas la transformation du programme pour générer des déclarations de mode est vraiment intéressante.

4.1 Compilation d'informations de contrôle.

4.1.1 Règle de calcul.

Le système de compilation d'informations de contrôle a été présenté comme un moyen d'améliorer des programmes pour lesquels la règle de calcul de prolog ne permet pas d'obtenir une exécution efficace. La solution proposée est de définir une règle de calcul idéale qui permette d'obtenir l'exécution souhaitée, puis de transformer ce programme en un programme prolog dont l'exécution est identique au programme original avec la règle de calcul idéale.

D'autres méthodes pour résoudre les problèmes provoqués par la règle de calcul de prolog ont été proposées [Naish 85a], [Naish 85b]. L'avantage de cette méthode-ci est de "compiler" la règle de calcul dans le nouveau programme.

a. Informations non compilées.

Le système présenté dans [De Schreye 87], ne permet pas compiler entièrement la règle de calcul.

Dans le programme compilé, les nouvelles clauses ont la forme
```
P(t1,...tn) :-
     Q1,
     ...,
     Qk,
     shuffle_c(t1,...,tn,L1',...,Lm'),
     Q(L1',...,Lm').
```

Lors de l'exécution du programme, la tête est toujours unifiée avec un prédicat de la forme P(L1,...,Ln). L1,...,Ln et L1',...,Lm' sont des listes dont tous les éléments ont une instanciation semblable (qui peut être décrite par le même atome_0) et telles que les éléments de Li (resp Li') ont une instanciation différente des éléments de Lj, i=/=j (resp Lj').

shuffle_c(L1,...,Ln,L1',...,Lm'), est introduit pour qu'après l'unification de la tête et l'exécution de Q1,...,Qk, les éléments de L1,...,Ln soient correctement rangés dans les listes L1',...,Lm'.

En choisissant bien le type des éléments de L1,..,Ln et L1',...,Lm', Shuffle_C(...). peut souvent être ignoré, par exemple quand il existe Li tel que Lj'=Li ou tel que Lj' est Li sans le premier élément. Par contre quand l'unification de la tête et l'exécution de Q1,...,Qk modifient l'instanciation des éléments de L1,...,Ln, alors shuffle_c(...) est nécessaire. Dans ce cas, shuffle_c(...) correspond à la partie de la règle de calcul qui n'a pu être compilée.

Un nouveau système qui tient compte des changements d'instanciation des éléments de L1,...,Ln et qui permettrait de supprimer complètement les prédicats shuffle_c est à l'étude.

b. Limites du système.

La méthode est basée sur une règle de calcul basée sur l'instanciation des prédicats. Il semble toutefois possible de compiler des programmes dont la règle de calcul est définie autrement. Par exemple pour compiler des programmes écrits en MU-PROLOG, FCP, PARLOG, tant que l'exécution ne dépend pas de données introduites lors de l'exécution.[Bruynooghe 86].

On remarque toutefois que la méthode ne permet de compiler des programmes que pour une directionnalité (Ex: sort/2 est compilé pour la directionnalité In(gr,var) Out(gr,gr). Cela entraîne que certains problèmes de contrôle ne peuvent être résolus.

exemple:

soit la procédure

```
perm([],[]).
perm(L,[H|T]) :-
     delete(H1,L,L1),
     perm(L1,T).
```

Pour la directionnalité In(gr,var) Out(gr,gr) delete doit s'exécuter avant perm.
Pour la directionnalité In(var,gr) Out(gr,gr) delete doit s'exécuter après perm.[Naih 85a].

Ce problème peut facilement être résolu en prolog en écrivant deux procédures, une pour chaque directionnalité, et en ajoutant des procédures de contrôle var(X), ground(X) pour sélectionner la bonne version.

En étendant les notations de façon à permettre d'indiquer que l'on ne connaît pas l'instanciation d'un terme (ex: remplacer les atomes_1, par des atomes_2), le système permet de compiler pareils programmes. On obtient à nouveau une clause pour chaque directionnalité et les procédures de contrôle var(X), ground(X) sont automatiquement ajoutées lors de l'élagage des branches superflues.

4.1.2 Système de tranformation.

La compilation d'informations de contrôle est présentée comme un moyen de redéfinir la règle de calcul d'un programme. Ce système est également présenté comme un système de transformation de programmes, similaire au système présenté dans [Burstall 77].

Dans un premier temps, je vais montrer que la méthode proposée correspond à un tel système de tranformation. J'indiquerai ensuite en quoi ce système de transformation est fondamentalement différent de celui proposé dans [Deville 87].

a. Similitudes à un système de transformation.

Les différentes phases de la compilation du programme peuvent être comparées à un système de transformation comme suit:
-la définition de la règle de calcul et la construction de l'arbre de résolution symbolique correspond au "dépliage" des prédicats,
-la construction des nouvelles clauses au "pliage".

Cette méthode n'utilise pas d'"Eureka" pour plier les prédicats.

L'analyse du fonctionnement du système confirme les similitudes avec un système de transformation.

Pour pouvoir transformer un programme, l'utilisateur doit fournir au système le texte du programme à transformer et des exemples de questions. La règle de calcul est construite par dialogue avec l'utilisateur. Pour chaque question exemple, le système demande à l'utilisateur le prédicat qui doit être sélectionné dans chaque noeud de l'arbre de résolution du programme pour la question exemple, ainsi que le mode de calcul du prédicat sélectionné. La cohérence de la règle de calcul est vérifiée au fur et à mesure, le système essaie de choisir le prédicat à l'aide de la partie de la règle de calcul déjà définie et n'interroge l'utilisateur que s'il ne possède pas assez d'informations.

On remarque que la règle de calcul est avant tout un élément interne qui permet de s'assurer que le programme est déplié de manière à pouvoir être replié ensuite. Le type et la définition de cette règle peuventt d'ailleurs être complètement ignorés de l'utilisateur.

b. Différences avec un système de transformation.

Je crois que la différence fondamentale entre le système de transformation proposé dans la compilation d'informations de contrôle, appelé S1, et le système proposé dans [Deville 87], appelé S2, est que les transformations de S1 sont basées sur l'exécution du programme, point de vue procédural, tandis que les transformations de S2 sont basées sur la définition du programme, point de vue déclaratif. Cette différence est surtout importante vis-à-vis de la correction des transformations.

Dans S1, la correction de la transformation se base sur l'équivalence entre l'arbre de résolution symbolique du programme original et l'arbre de résolution symbolique du programme transformé. Déterminer si les deux programmes sont logiquement équivalents est un problème qui n'est toujours pas résolu.

Les arbres de résolution symbolique des deux programmes sont identiques par construction. Cependant, l'arbre de résolution symbolique du programme de départ est généralement infini et la transformation se base sur un sous-arbre fini de celui-ci. Il est alors nécessaire de prouver que ce sous-arbre fini est suffisamment grand pour que la transformation soit correcte. Cette démonstration est malheureusement difficilement automatisable. Dans le système présenté, l'équivalence entre les deux programmes est vérifiée en testant s'ils donnent les même réponses pour les questions exemples fournies au système.

Dans S2, la correction de la transformation est obtenue par l'équivalence logique des deux programmes.

La méthode de transformation garantit cette correction en ne permettant que de passer d'un programme à un autre programme qui lui est logiquement équivalent.

Les deux systèmes peuvent également être comparés au point de vue intervention nécessaire de l'utilisateur.

Dans S1, l'utilisateur doit uniquement indiquer comment déplier le programme et il est d'ailleurs assisté dans cette tâche. Le reste de la transformation se fait normalement sans lui.

Dans le système S2, l'utilisateur doit indiquer comment déplier et plier le programme. Souvent une définition auxiliaire, appelée Eureka, et des propriétés de la spécification des procédures sont également nécessaires pour transformer le programme. L'Eureka permet d'apporter des informations (de la sémantique) supplémentaires au programme et est souvent la base de transformations très puissantes qui dépassent la simple exécution de celui-ci. Dans S2, des heuristiques pour pouvoir trouver certains Eurekas sont proposées et une base de connaissance peut servir pour obtenir les propriétés nécessaires des procédures.

4.2 Génération de déclarations de mode.

4.2.1 Originalité du programme.

Plusieurs recherches ont déja été faites sur la génération automatique d'informations de contrôle, [Mellish 81], [Debray 86b]. Le but de ces recherches est d'obtenir les différentes instanciations possibles d'un prédicat au moment de l'appel et de généraliser cette information pour pouvoir générer des déclarations de mode.

Dans mon cas, cette information est obtenue à partir de l'arbre de résolution symbolique à partir duquel le programme est compilé.

L'originalité du programme réalisé est de transformer le programme pour lequel on veut faire une déclaration de mode, quand l'information disponible sur l'instanciation d'un prédicat ne permet que de déclarer le mode d'un argument ? (inconnu) alors que l'on dispose d'informations qui permettent de faire mieux.

Après avoir rappelé le type de transformations réalisées, je verrai en quoi ces transformations sont intéressantes pour différents types d'optimisation possibles grâce aux déclarations de mode.

4.2.2 Rappel.

Soit P(t1,..,tn) un prédicat et Ag un atome_2 qui décrit l'instanciation de P/n au moment de sa sélection par la règle de calcul, le programme réalisé permet d'effectuer les transformations suivantes :

- **transformation 1**. Si dans Ag l'instanciation de ti est décrite par un terme_2 de la forme f(a1,...,am), alors on peut uniquement déclarer qu'on ne connaît pas l'instanciation de ti. Le programme est transformé pour que dans toutes occurrences de P/n le ième argument soit remplacé par m arguments $t_{i1},...,t_{im}$.
- **transformation 2**. Si Ag permet de déterminer que lors de l'appel de P/n certaines variables sont toujours liées, il est possible de transformer le programme pour en éliminer une.

4.2.3 Optimisations réalisées.

Dans un compilateur, les déclarations de mode sont utilisées pour réaliser différents types d'optimisation. Je vais en citer trois [Debray 86b], puis je verrai cas par cas, s'il est utile de transformer le programme pour ce type d'optimisation.

Les déclarations de mode permettent les trois types d'optimisation suivants :

- dans des implémentations avec partage de structures (structure sharing), les déclarations de mode permettent de diminuer l'espace mémoire utilisé en allouant plus de variables sur le Local Stack.
- les déclarations de mode permettent d'accélérer l'unification en utilisant des routines spécialisées.
- il est également possible d'utiliser les déclarations de mode pour inférer le déterminisme ou la fonctionnalité de certains prédicats.

a. Implémentation "structure sharing".

Pour ne pas entrer dans les détails d'implémentation de compilateur prolog, je vais seulement citer dans quels cas l'utilisation de déclarations de mode permet de gagner de l'espace mémoire.

Exemple :

Si la tête d'une clause a la forme p(...f(X,Y),...,), alors les variables X,Y doivent être mises sur le "global stack". Par contre si on sait que f(X,Y) ne sera jamais unifié avec une variable, X et Y peuvent être mises sur le "local stack". [Bruynooghe 82].

Placer les variables dans le "local stack" plutôt que dans le "global stack" permet de gagner de la place mémoire.

Dans ce cas-ci, la transformation 1 est intéressante pour tous les arguments qui sont de la forme f(t1,...,tn) si le terme_2 qui décrit l'instanciation de cet argument contient des variables Gi, i∈N.

Si on reprend l'exemple, et que l'on suppose que l'instanciation de f(X,Y) est décrite par f(G1,V2), p/n va être transformé en un prédicat p(...,X,Y,...). Dans la déclaration de mode il sera possible de déclarer X comme Input et seul Y devra être mise sur le "global stack".

b. Unification.

Pour expliquer le genre d'optimisation qui peut être réalisé ici prenons l'exemple de la procédure append/3, [Mellish 81] :

```
append([],L,L).
append([H|T],L,[H|T1]L) :-
            append(T,L,L1).
```

Cette procédure peut être utilisée pour répondre à des questions telles que:

```
?- append([a,b,c],[d,e],X).
?- append(X,Y,[a,b,c])
```

Pour la seconde clause et des questions du premier type le code généré pour unifier le premier argument ressemble à :

-vérifier que le premier argument de la question est une liste,
-instancier la variable H au premier élément de la liste,
-instancier la variable T au reste de la liste.

Pour des questions du second type le code généré ressemble à :

-construire une liste dont la tête a la même valeur que H et la queue la même valeur que T.
-instancier la variable X à cette liste.

Sans information supplémentaire, du code doit être généré pour faire face aux deux éventualités.

Dans ce cas de la procédure append/3, la transformation ne permet pas de résoudre le problème car elle n'est normalement pas possible. En effet, si le premier paramètre est décrit comme étant Gi, Vi ou Ni, i∈N, la transformation n'est pas nécessaire, si le premier argument est décrit comme une liste d'au moins un élément, cela veut dire qu'aucun appel à append/3 ne s'unifie avec la première clause, ce qui n'est pas possible si le programme est correct.

Par contre, quand dans une tête de clause un argument est de la forme, f(t1,...,tn) et que l'instanciation de cet argument est décrite par un terme_2 T2=f(a1,...,an) tel que T2 ne contient pas que des variables_2 Ni, la tranformation 1, permet d'optimiser le code généré.

exemple:

Soit deux têtes de clause, p(...,f(X,[]),...) et p(...,f(X,[H|T]),...) dont l'instanciation est décrite par p(....,f(V1,G3),...). En transformant la procédure, on obtient les deux prédicats p(...,X,[],...) et p(...,X,[H|T],...) pour lesquels on peut faire la déclaration de mode p(...,o,i,...).

Il est aussi intéressant d'indiquer que dans certains cas, des transformations peuvent réduire les performances.

Si j'ai une clause :

```
p(...,X,...) :-
     ...,
     q(...,X,...),
     ...
```

où X n'apparaît qu'aux deux endroits mentionnés.

Si dans l'atome_2 qui décrit l'instanciation de p, l'instanciation de X est décrite par un terme_2 f(a1,...,an), la transformation 1 est toujours possible. On obtient une clause de la forme :

```
p(...,X1,...,Xn,...) :-
     ...,
     q(...,X1,...,Xn,...),
     ...
```

Maintenant au lieu de devoir unifier X avec un terme dont on ne connaît pas l'instanciation, on doit unifier n variables X1,...,Xn ce qui peut entraîner une perte de performance.

Par contre, la transformation peut être avantageuse si X doit de toute manière être remplacé par X1,...,Xn dans q(...). En effet si on ne tranforme pas p(...) on obtient une clause :

```
p(...,X,...) :-
     ...,
     X=f(X1,...,Xn)
     q(...,X1,...,Xn,...),
     ...
```

et les n unifications doivent quand même être faites.

Il faut toutefois se méfier de ce cas avantageux. Comme la transformation d'un prédicat doit se faire de manière cohérente dans tout le programme, il se peut que la transformation de p(...) finisse par sevir de justificatif à une transformation de ce genre dans d'autres clauses du programme.

La transformation 2 semble avantageuse au point de vue unification puisqu'elle permet de supprimer un certain nombre de termes. Il faut à nouveau être prudent. Pour pouvoir faire cette transformation, il est souvent nécessaire de remplacer des variables par des termes f(....). Le gain obtenu en supprimant un terme peut être annulé par l'augmentation du nombre d'unifications nécessaires comme expliqué dans l'exemple précédent.

c. Déterminisme ou fonctionnalité.

Les déclarations de mode peuvent également être utilisées pour inférer le déterminisme ou la fonctionnalité de certains prédicats. Ces propriétés peuvent à leur tour servir à détecter la récursion terminale et/ou à transformer un programme, par exemple insérer des "cut" pour contrôler le backtracking. [Debray 86a].

Un prédicat est fonctionnel par rapport à un mode M, si les clauses qui le définissent sont mutuellement exclusive deux à deux et chaque clause est fonctionnelle par rapport au mode M.

1. Exclusion mutuelle.

La transformation 1 peut aider à déterminer plus précisément si deux clauses sont mutuellement exclusives:

Si dans la tête des deux clauses, un argument est de la forme, f(t1,...,tn) et que l'instanciation de cet argument est décrite par un terme_2 T2=f(a1,...,an) tel que T2 contient des variables_2 Gi, la transformation 1 peut permettre de déterminer plus précisément si ces clauses sont mutuellement exclusives.

exemple:

deux clauses

```
p(...f(X,[]),...) :- ....
p(...,f(X,[H|T]),...) :- ....
```

l'instanciation de p est décrite par

```
p(....,f(V1,G3),...)
```

la transformation 1 permet d'obtenir les deux nouvelles clauses :

```
p(...,X,[],...) :- ....
p(...,X,[H|T],...) :- ....
```

dont la déclaration de mode est:

```
p(...,o,i,...).
```

Il est maintenant possible de déterminer que les deux nouvelles clauses sont mutuellement exclusives, puisqu' un terme de base ne peut s'unifer à [] et à [H|T].

Ce cas-ci est à peu près identique à celui présenté pour l'unification. Dans ce cas-ci, il est toutefois nécessaire de pouvoir faire une déclaration de mode "i". la transformation n'est évidemment utile que s'il n'est pas possible de déterminer l'exclusion mutuelle des deux clauses autrement.

2. Fonctionnalité.

La fonctionnalité d'une clause est déterminée par la fonctionnalité des litéraux de cette clause. La fonctionnalité de ces litéraux est elle déterminée à l'aide d'une proposition basée sur des déclarations de mode et des dépendances fonctionnelles entre les arguments de ces litéraux.

La **dépendance fonctionnelle** entre les arguments d'un prédicat est définie comme suit: Soit un prédicat p(x1,...,xn), s'il existe des sous-ensembles de ces arguments, U et V tels qu'une occurrence de base des arguments de U détermine l'instanciation des arguments de V alors **V dépend fonctionnellement de U**, noté $\mathbb{U} \rightarrow \mathbb{V}$.

proposition:

Un litéral p(x1,...,xn) est fonctionnel par rapport à une déclaration de mode M, si dans tous les appels qui respectent M, il existe U,V,W des sous-ensembles de {x1,...,xn} tels que U u V u W ={x1,...,xn}, U -> V, dans tous les appels qui respectent M, les éléments de U sont des termes de base, W contient des variables qui n'apparaissent pas dans un autre litéral de la clause.

La transformation 1 peut être utile si elle permet également d'obtenir des dépendances fonctionnelles plus précises. L'article ne précise malheureusement pas comment ces dépendances fonctionnelles sont obtenues.

exemple :

Soit un prédicat p(...,X,...,Y,...) dont l'instanciation est décrite par l'atome_2 p(...,G1,...,f(V2,V3),...). La transformation 1 permet d'obtenir le nouveau prédicat p(...,X,...,Y1,Y2,...). Rien n'empêche qu'il y ait par exemple dépendance fonctionnelle entre X et Y1, X -> Y1, ce qui permettrait de déterminer plus précisément si p est fonctionnel ou pas.

d. Conclusion.

On vient de voir dans quels cas les transformations effectuées permettent d'optimiser le programme. On a aussi remarqué que dans certains cas, ces transformations peuvent diminuer les performances du programme.

Si l'on ne considère que les deux premières optimisations, il n'est utile de transformer un argument d'un prédicat que si dans la tête des clauses qui définissent ce prédicat, cet argument a la forme f(t1,...,tn) et est décrit par un terme_2 f(a1,...,a2) qui ne contient pas que des variables Ni.

4.2.4 Construction du programme.

a. Arbres de résolution symbolique.

L'instanciation des différents prédicats du programme, pour pouvoir générer les déclarations de mode est obtenue à partir de l'arbre de résolution symbolique à partir duquel le programme a été compilé. Cela entraîne qu'il faut savoir comment les prédicats du programme sont construits à partir de l'arbre de résolution symbolique, et donc comprendre comment le système de compilation fonctionne. Il aurait été plus simple de travailler directement à partir de la trace du programme compilé, malheureusement le système de compilation ne permettait pas d'obtenir facilement cet arbre symbolique, principalement à cause des prédicats shuffle_∞. Il aurait évidemment été possible de réaliser un programme qui permette d'obtenir directement l'instanciation des prédicats du nouveau programme, par exemple par interprétation abstraite [Debray 86b], mais cela est un problème non trival, et dans le cas présent il me semblait nettement plus facile d'obtenir l'information à partir de l'arbre de résolution symbolique de départ.

b. Atomes d'instanciation.

Dans l'arbre de résolution symbolique, l'instanciation d'une variable est décrite par une variable_1 Vi et un terme de base par une variable_1 Gi. De cette manière, la construction de l'arbre de résolution symbolique peut être expliquée de manière analogue à la construction d'un arbre de résolution SLD-Tree. Par contre, représenter des termes de base par des variables est assez déconcertant au début, de plus comme les variables_1 Gi représentent des termes de base, il est nécessaire d'adapter les règles d'unification aux variables Gi et Vi.

Pour pouvoir décrire la forme générale d'un prédicat, des variables Ni (pour n'importe quelle instanciation "any") ont été introduites. On obtient ainsi des atomes d'instanciation_2. L'avantage de cette représentation est qu'un atome_1 est un cas particulier d'un atome_2, la fonction gen(A1,A2) est ainsi plus facile à définir.

Représenter les termes de base par des variables Gi et les termes dont on ne connaît pas l'instanciation par des variables Ni permet à nouveau d'expliquer facilement les principes de transformation en termes d'unification et de quelques fonctions simples telles qu'aplatir un terme, enlever des doubles.

Par contre, pour construire le programme il est seulement nécessaire de savoir s'il s'agit d'un terme de base ou d'un terme dont on ne connaît pas l'instanciation. Il suffit alors d'un terme_2 pour représenter tous les termes de base et d'un terme_2 pour tous les termes dont on ne connaît pas l'instanciation. En fait pour rester cohérent avec les principes de transformation, toutes les variables Ni et Gi sont considérées comme différentes. Cela ne pose pas de problème, car si l'instanciation d'un prédicat est décrite par un atome_2 où les variables_2 Gi peuvent être identiques, (par définition

de la fonction gen(A1,A2) les variables Ni sont toutes différentes), l'instanciation de ce prédicat est également décrite par le même atome_2 où toutes les variables_2 Gi sont considérées comme différentes. Cela entraîne juste une perte d'information sans importance puisqu'on ne se sert pas de cette information pour générer les déclarations de mode.

Représenter les variables_2 par des variables prolog n'est pas très intéressant. Cela ne permet pas de reconnaître facilement une variable Vi d'une variable Gi ou Ni, et de reconnaître les variables Vi identiques. Les variables_2 Gi et Ni ont été représentée par deux constantes g et any, les variables Vi par des constantes vi, (i ∈ N). Cette représentation est "cachée" via le module "termes prolog, termes_0 et termes_2".

c. Construction des algorithmes.

Le programme à réaliser se prêtait bien à l'utilisation de la méthodologie. La plupart des procédures, peuvent être spécifiées de manière déclarative, n'ont pas d'effet de bord et ne dépendent pas de préconditions de l'environnement.

Dans certains cas, le programme a été volontairement simplifié. Par exemple dans le module "programme", plutôt que de considérer que le programme à transformer se trouve dans un fichier, j'ai décidé de le représenter par une liste de clauses. Cela permet d'obtenir des procédures prolog dont l'aspect purement procédural est moins important. Des préconditions ont été ajoutées et les usages possibles (directionnalité) volontairement limités pour permettre de changer facilement de représentation.

La construction du programme repose autour d'un petit nombre de procédures :

-une procédure permet d'obtenir l'instanciation d'un prédicat à partir d'un noeud de l'arbre de résolution symbolique.
-une seconde procédure permet de généraliser deux atomes_2.
-on a déja vu que l'unification de prolog ne peut être utilisée, il était donc nécessaire de contruire une procédure d'unification. Etant donné que les variables_2 ne sont pas des variables prolog, une procédure pour unifier un prédicat et un atome_2 est nécessaire. Cette procédure était la plus complexe à construire. Une autre procédure pour calculer T θ l'instance de T en fonction de la substitution θ a également dû être construite.
-finalement les différentes fonctions définies pour transformer un prédicat, l'aplatir et enlever les doubles, sont calculées par une seule procédure.

L'architecture du programme, découpe en niveaux et en modules, a un double intérêt. De manière assez classique, elle permet de rendre le programme indépendant de choix de représentation des données et d'introduire des niveaux d'abstraction. Cela peut également être intéressant pour cacher les aspects non logiques du programme, procédures qui dépendent de préconditions de l'environnement et/ou à effets de bord.

d. Extension du programme.

La transformation de programme pour générer des déclarations de mode est particulièrement intéressante pour les programmes obtenus par la compilation d'informations de contrôle. Dans ces programmes, lors de la sélection d'un nouveau prédicat, tous les arguments de ce prédicat sont des listes de termes qui ont la même instanciation. Comme ces termes ne sont presque jamais de base, il n'est pratiquement jamais possible de faire une déclaration de mode.

Sous sa forme actuelle, le programme réalise toutes les transformations possibles, ce qui peut donner lieu à des situations assez étranges. Par exemple, si on réalise toutes les tranformations possibles sur le programme Lucky_new,(voir annexe), le programme transformé avec les déclarations de mode est plus lent que le programme original sans déclaration de mode. Cela confirme ce qu'on a dit à propos des optimisations réalisées pour l'unification : dans certains cas la transformation peut diminuer les performances du programme.

Il est donc souhaitable de réaliser une extension du programme qui irait d'abord consulter le programme pour déterminer dans quels cas une transformation est utile. Cela peut de faire en remplaçant, dans l'atome_2 qui décrit de manière générale l'instanciation d'un prédicat, les arguments de la forme f(a1,...,an) pour lesquels la transformation 1 n'est pas souhaitable par une variable Ni.

Chapitre 5
Evaluation de la méthodologie

Ce chapitre est basé sur mon expérience d'utilisation de la méthodologie qui est principalement constituée de la construction du programme de génération de déclarations de mode.

Les différentes phases de la construction d'une procédure vont être examinées une à une suivant la manière dont elles ont été présentées au chapitre 1. Pour chaque point la manière dont la méthodologie a été utilisée sera présentée ainsi que quelques modifications ou extensions.

5.1 Spécification.

Le but de la spécification est double. Lors de la construction d'une procédure, la procédure doit être correcte par rapport à sa spécification. Il est donc nécessaire de vérifier que toutes les parties de la spécification sont correctes. Lorqu'on se sert d'un sous-problème lors de la construction d'une procédure, la spécification du sous-problème est indispensable pour prouver que la procédure est correcte par rapport à sa spécification. Certaines informations facilitent également la construction de la procédure logique.

5.1.1 Types et autres préconditions.

Les déclarations de type sont utiles à plusieurs points de vue. Lors de la construction de l'algorithme logique, elles permettent de définir une relation bien fondée sur le paramètre d'induction. Elles permettent également d'exprimer plus facilement la relation entre les différents arguments de la procédure qui doit uniquement être vraie pour des arguments qui respectent ces déclarations de type et les autres préconditions. De ce fait, moins de cas doivent être pris en considération lors de la contruction de l'algorithme logique.

Lors de la dérivation d'une procédure logique les déclarations de type constituent un ensemble de contraintes qu'il faut vérifier, type des sous-problèmes utilisés et post-conditions.

5.1.2 Relation.

La relation est la partie centrale de la spécification. La correction de l'algorithme logique et de la procédure est établie par équivalence avec cette relation.

La procédure détermine si une relation R existe entre les paramètres A1,...,An de la procédure. Par relation, on entend un ensemble de tuples de base <a1,...,an>. Quand les paramètres de la procédure ne sont pas des termes de base, la procédure détermine s'il existe une instance de base des paramètres telle que <A1θ,...,Anθ> ∈ relation.

Dans certains cas, la relation ne peut être définie sur un ensemble de tuples de base, car cela n'a pas de sens. Par exemple si l'on veut spécifier la primitive Prolog var(X), dire que "la procédure détermine si X est une variable" n'a pas de sens. Dans ce cas précis, il est possible de ne pas définir de relation du tout, l'effet de la procédure peut être entièrement déterminé par la directionnalité. Je préfère toutefois toujours donner une relation entre les paramètres de la procédure car je trouve que cela aide à mieux comprendre la spécification. La relation est alors un ensemble de tuples <a1,...,an> avec ai des termes prolog.

La spécification de var(X) devient:

procédure var(X)

Soit
X un terme,

Cette procédure détermine si

X est une variable.(Extra logique).

Directionnalité.

In(var) Out(var). <1,1>.
In(novar) <0,0>.

Remarques:

-Comme mentionné plus haut, on remarque que l'effet de la procédure est entièrement déterminé par la directionnalité.
-Pour la directionnalité In(novar) ça ne sert à rien d'indiquer une partie Out(...) puisque dans ce cas la procédure ne réussit jamais (Max=0).
-Il est nécessaire d'indiquer que la relation n'est pas établie sur des termes de base. Dans ce cas-ci, cela est mentionné en disant qu'elle est "Extra logique".

5.1.3 Directionnalité.

La partie In(..) de la directionnalité permet de simplifier la construction de la procédure. Comme cela a déjà été dit lors de la présentation de la méthodologie, choisir un paramètre d'induction de base permet de dériver plus facilement la procédure logique à partir de l'algorithme logique. Il semble également que les procédures logiques dérivées d'algorithmes logiques construits avec le paramètre d'induction de base sont plus performantes.

Dans le programme réalisé, j'ai toujours essayé de choisir le paramètre d'induction de base. Malheureusement dans la plupart des procédures, les arguments qui étaient de bons candidats comme paramètres d'induction étaient rarement de base. Par exemple la procédure principale ne peut se limiter à transformer des programmes qui sont de base si on veut qu'elle soit utile.

Je crois que pour le type de programme que j'ai écrit, il est plus opportun de parler de paramètre suffisamment instancié. Son instanciation doit être telle qu'il soit possible de prouver que le paramètre correspondant dans un appel récursif est plus petit selon la relation bien fondée.

exemple:

si je prend la clause

```
p([H|T],....) <-
          p(T,.....).
```

si la procédure est appelée avec une liste de la forme [X1,X2|T] comme premier argument, il n'est pas possible de prouver que T < [H|T]. Par contre si elle est appelée avec une liste de la forme [X1,toto,f(Y1,Y2)] alors T < [H|T].

On remarque qu'il est seulement nécessaire que la procédure soit appelée avec une liste de longueur fixe comme premier argument.

Dans la méthodologie trois formes, var, gr, ngv, ont été retenues pour exprimer l'instanciation d'un paramètre quand on décrit la directionnalité d'une procédure. Je pense qu'il peut être utile d'en ajouter deux autres.

La forme ngv (pour neither ground nor var), ni variable ni terme de base, n'est pas assez précise pour distinguer une liste de longueur finie dont les éléments ne sont pas tous des termes de base d'une liste de longueur infinie.

Dans la spécification du programme, la solution suivante a été adoptée: pour chaque structure de données qui pouvait poser problème, une nouvelle forme a été introduite pour exprimer l'instanciation d'un paramètre de ce type. Par exemple l'instanciation d'un programme, d'une substitution, d'une liste, est respectivement décrite par P(Inst), S(Inst), L(Inst) où Inst décrit l'instanciation des clauses du programme, des éléments de la liste ou de la substitution. On remarque que L(gr), S(gr), P(gr) sont équivalents à gr.

Il est possible d'utiliser une seule notation I(Inst) où Inst a la même signification que précédemment et I, (pour input), indique que le paramètre est une donnée de longueur finie.

Une autre extention possible est de permettre d'exprimer dans la partie Out(...) de la directionnalité, que l'instanciation d'un paramètre est la même que dans la partie In(...). Cette extension n'est nécessaire que quand la forme du paramètre est ngv dans la partie In(...). En effet, si la forme du paramètre est gr elle ne peut pas être changée, si elle est var et est modifiée alors il suffit d'indiquer qu'elle est novar en sortie, par contre si elle est ngv elle peut être modifiée et rester ngv.

exemple:

dans la procédure unify(T1,T2,M,T) qui détermine si M est le mgu de T1 et T2 et T = T1 M, il est intéressant d'indiquer que si T1 et T2 ne sont pas des termes de base alors ils ne sont pas modifiés. On obtient la directionnalité In(any,any,var,var) Out(nc,nc,S(ngv),any), où nc (not changed) indique que le paramètre n'est pas modifié.

Cette extension permet également d'exprimer plus simplement la directionnalité d'une procédure. Si une procédure est correcte quelle que soit la forme d'un paramètre et que ce paramètre n'est pas modifié, la déclaration In(any,...) Out(any,...) peut permettre d'exprimer facilement cette directionnalité. Pour prouver la correction d'une autre procédure il est parfois nécessaire de savoir qu'un argument qui est var reste var, ngv reste ngv. Dans ce cas, il est nécessaire de faire les déclarations :
In(var,...) Out(var,...),
In(gr,...) Out(gr,..),
In(ngv,...), Out(ngv,...)
alors qu'avec la notation nc une seule déclaration suffit :
In(any,...) Out(nc,...).

5.2 Construction en deux phases.

Je pense qu'un des points les plus importants de la méthodologie est de diviser la construction d'une procédure en deux phases, construction d'un algorithme logique et dérivation d'une procédure prolog. Cela permet de tirer parti de la sémantique procédurale et déclarative de prolog. La correction d'une procédure est également définie par rapport à ces deux aspects.

La partie la plus créative de la construction est certainement l'obtention d'un algorithme logique correct par rapport à une spécification. En fait on ne s'intéresse dans la première phase qu'à la correction par rapport aux types, aux autres préconditions et à la relation.

Ensuite, une procédure logique est dérivée et l'on vérifie qu'elle est correcte du point de vue procédural. Finalement diverses optimisations (insérer des cut, etc ...) sont réalisées tout en préservant la correction. Le principe retenu est qu'il est plus facile d'optimiser une procédure correcte que de corriger une procédure efficace.

Lors de la construction d'une procédure, la séparation entre ces deux phases n'est pas aussi marquée. On a tendance à faire beaucoup plus de choses simultanément.

Les différentes phases de construction ressemblent plutôt à :

-choisir un paramètre d'induction Xj,
-définir une relation bien fondée sur le type de Xj,
-construire les Ci à partir des différentes formes structurelles de Xj. Un des Ci doit couvrir le cas minimal et toute forme structurelle de Xj doit être couverte.

Les trois premières phases ne changent pas,

-la phase suivante consiste à construire les Fi de manière à ce que Ci & Fi soient une condition nécessaire et suffisante pour qu'une instance de base des paramètres qui respecte les préconditions fasse partie de la relation quand la forme structurelle de Xj est décrite par Ci.

Lors de la construction des Fi, je vérifie également qu'il est possible de dériver une clause telle que les préconditions de tous les sous-problèmes soient respectées et telle qu'il soit possible de placer l'appel récursif en dernière position (si ce n'est pas possible, j'envisage de généraliser la clause).

Le choix de la relation bien fondée et des Ci, et la construction des Fi sont fort dépendants, ces étapes sont plutôt réalisées en parallèle.

La dérivation d'une procédure prolog devient :

-tranformer l'algorithme sous forme de clauses,
-si le paramètre d'induction n'est pas de base et que le nombre de substitutions réponses est fini, vérifier que la procédure se termine,
-vérifier que les postconditions (types, autres préconditions et directionnalité) sont bien vérifiées.
-optimiser la procédure.

Ce regroupement des différentes phases de construction se marque sur la manière d'écrire les algorithmes logiques.

exemple : J'écris souvent de la manière suivante :

```
p(x1,...,xk) <=> C1 & F1
          V     .....
          V     Ci & Fi1
          V     Ci & ...
          V     Ci & Fim
          V     ......
          V     Cn & Fn.
```

un algorithme qui selon les conventions de la méthodologie devrait être écrit :

```
p(x1,...,xk) <=>  C1 & F1
          V     .....
          V      Ci & (Fi1 V ...V Fim)
          V      .....
          V      Cm & Fm.
```

Cela ne pose évidemment pas de problème puisque les deux algorithmes sont logiquement équivalents mais me semble révélateur de la manière de procéder.

Les algorithmes logiques tels que je les écris sont en fait déjà presque sous forme de clausale.

De plus pour chaque Ci & Fij, les différents litéraux sont le plus souvent écrits dans un ordre qui permet de dériver une procédure logique correcte sans permuter les litéraux. Il me semble qu'ordonner les litéraux de cette façon permet de comprendre plus facilement l'algorithme.

Je pense que découper la construction d'une procédure prolog en deux phases est néanmoins très pratique.

Cela permet :

-d'expliquer plus facilement la méthodologie,
-de marquer que l'on désire une procédure correcte du point de vue logique et du point de vue déclaratif,
-dans la documentation du programme l'algorithme logique est souvent plus clair que la procédure prolog correspondante qui a souvent été optimisée en ajoutant des cuts etc ...

5.3 Algorithmes logiques.

5.3.1 Vérification de types.

Quand on construit un algorithme logique on n'utilise pas uniquement des litéraux de base mais principalement des litéraux existentiellement quantifiés, (étant donné nos conventions ces quantificateurs sont implicitement présents). Normalement le critère de correction impose que les préconditions de ces litéraux soient explicitement vérifiées en ajoutant par exemple des litéraux pour vérifier ces préconditions.

Lorsque l'on vérifie la correction de la procédure logique dérivée, et plus précisément lorsqu'on vérifie les types, ces litéraux peuvent le plus souvent être supprimés. Soit que le paramètre pour lequel on a ajouté un litéral de vérification de type est une variable lors de la sélection de ce litéral et donc la précondition est trivialement vérifiée, soit que l'exécution des litéraux qui précèdent garantisse que les préconditions sont respectées.

Comme je viens de l'expliquer, j'ai un peu adapté les deux phases de construction de l'algorithme. J'ai donc également décidé de ne pas ajouter systématiquement des litéraux pour vérifier les préconditions des variables existentiellement quantifiées, mais uniquement s'ils sont nécessaires à la correction de la procédure prolog dérivée.

5.3.2 Construction des algorithmes.

Dans la méthodologie, une méthode pour construire des algorithmes logiques corrects par construction est présentée. Trois types de généralisation, (structurelle, état de calcul descendante et ascendante) sont également présentés. Lors de la construction du programme, la majorité des algorithmes construits par induction ont été généralisés et les trois types de généralisations ont été utilisés. Dans certains cas, la généralisation n'était pas indispensable pour pouvoir construire l'algorithme, mais elle permettait soit de construire l'algorithme plus facilement, soit d'obtenir une procédure prolog plus efficace. Dans certains cas, une autre généralisation aurait pu être choisie, dans d'autres la procédure a été généralisée deux fois.

Je vais illustrer ces propos à l'aide d'un exemple où la procédure a été généralisée deux fois pour obtenir la version définitive.

5.3.2.1 Exemple.

Dans le chapitre 3, j'ai présenté la construction de la procédure fl_ndv/5, qui permet de calculer quelques fonctions de base nécessaires lors de la transformation d'un programme pour générer des déclarations de mode. Je vais examiner le processus de construction d'un algorithme logique sur une version simplifiée de cette procédure.

Spécification.

Soit un terme de base T=f(t1,...,tn) Tfnd, le terme T aplati sans double, noté Tfnd=Fnd(T), est défini comme suit:
-si ∃ ti tel que ti a la forme g(ti1,...,tim), alors
 T'=f(t1,...,ti-1,ti1,...,tim,ti+1,...,tn) et Tfnd=Fnd(T').
-si ∃ ti,tj tel que ti=tj, i<j, alors
 T'=f(t1,...,tj-1,tj+1,...,tn) et Tfnd=Fnd(T').
-sinon Tfnd=T.

procédure fnd(T,T1).

Soit
T,T1, des termes de la forme f(t1,..,tn).

Cette procédure détermine si

T1 = Fnd(T).

Construction.

Cette procédure peut être construite de multiple manières. La plus simple est certainement d'aplatir T et d'enlever les doubles séparément.

Si on veut faire les deux opérations en même temps, le plus simple est de construire une procédure fnd_L/2 pour transformer la liste des arguments de T. On obtient l'algorithme :

```
fnd(T,T1) <=>
     T=..[F|L] & fnd_L(L,L1) & T1=[F|L].
```

Spécifions et construisons la procédure fnd_L(L,L1).

procédure fnd_L(L,Lf)

Soit
L,Lf des listes de termes.

Cette procédure détermine si

Lf est la liste des arguments de fnd(L).

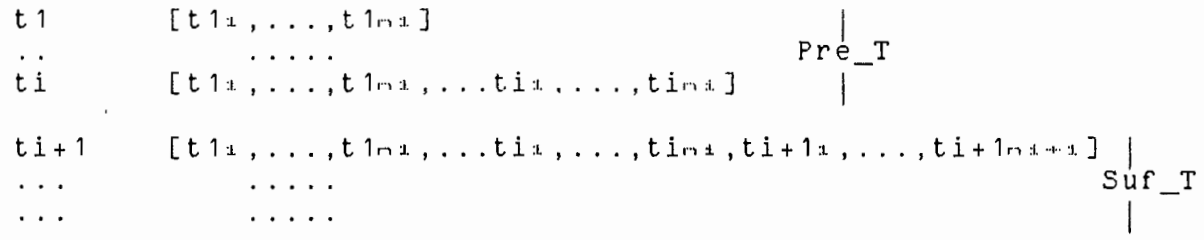
Cet algorithme peut être construit sans être généralisé, par induction sur L, on obtient l'algorithme :

```
fnd_L(L,Lf) <=>
     L=[] & Lf=[]

  V  L=[H|T] & compound(H) & H=..[F|Lh] &
     append(Lh,T,L1) & fnd_L(L1,Lf),

  V  L=[H|T] & ¬ compound(H) & fnd_L(T,Tf) &
     member(H,Tf) & Lf=Tf,

  V  L=[H|T] & ¬ compound(H) & fnd_L(T,Tf) &
     ¬ member(H,Tf) & Lf=[H|TF].
```

On remarque, que dans la procédure logique dérivée, il n'est pas possible de placer l'appel récursif en dernière position quand le premier élément de la liste n'est pas une structure. De plus, les deux opérations, aplatir les éléments et enlever les doubles, ne sont pas vraiment faites en même temps.

Généralisons la procédure pour obtenir un meilleur algorithme logique.

Si L=[t1,...,tn], on peut représenter un état de calcul par :



avec ni ≥ 0. ni = 0 quand ti ne contient que des termes qui existent déja dans [t1₁,..,t1ₘ₁,...ti-1₁,...,ti-1ₘᵢ₋₁].

Les informations nécessaires à propos de Pre_T pour changer d'état de calcul, réduire Suf_T, sont les éléments de la liste [t11,...,t1n1,...ti1,...,tini].

Construisons l'algorithme logique de la procédure généralisée dont voici la spécification.

<u>procédure</u> fnd_g(L,Lf,Ld).

Soit
L,Lf,Ld des listes de termes,

<u>Cette procédure détermine si</u>

L1 est la liste des arguments de fnd(L). Lf est L1 sans les termes qui apparaissent dans Ld.

Cette généralisation est une généralisation de fnd_L/2 : fnd_L(L,Lf) <=> fnd_g(L,Lf,[]).

L'algorithme peut être construit par induction sur L :

```
fnd_g(L,Lf,Ld) <=>
     L=[] et Lf=[],

  V  L=[H|T] & compound(H) & H=..[F|Lh] &
     append(Lh,T,Nt) & fnd_g(Nt,Lf,Ld).

  V  L=[H|T] & ¬ compound(H) & member(H,Ld) &
     fnd_g(T,Lf,Ld),

  V  L=[H|T] & ¬ compound(H) & ¬ member(H,Ld) &
     Ld1=[H|Ld] & Lf=[H|Lf1] & fnd_g(T,Lf1,Ld1).
```

Lf et Ld sont deux listes à peu près identiques. En fait Lf et Ld contiennent les mêmes éléments, (Ld est Lf inversée). La construction de ces deux listes est par contre assez différente. Lf est construite de manière descendante, on en cherche le premier élément et le reste de la liste est

construit récursivement (ce qui permet d'obtenir les éléments dans l'ordre désiré). Ld est construite de manière ascendante, on connaît déja les i-1 premiers éléments et on y ajoute le $i^{ème}$ (ce qui permet de connaître les i-1 premiers éléments quand on veut changer d'état de calcul).

En généralisant une seconde fois la procédure, il est possible d'obtenir une version où l'on n'est pas obligé de concaténer des listes lorsque le premier élément de L est une structure. Deux généralisations différentes qui permettent d'obtenir ce résultat sont proposées.

procédure fnd_g2(L,Lf,Lfint,Ld,Ldint).

Soit
L,Lf,Lfint,Ld,Ldint des listes.

Cette procédure détermine si

L1 est la liste des arguments de fnd(L) sans les termes qui apparaissent dans Ldint.
Lf est la concaténation de L1 et Lfint.
Ld est la liste des éléments de Ldint et de L1.

La procédure est construite par induction sur L:

```
fnd_g2(L,Lf,Lfint,Ld,Ldint) <=>
     L=[] & Lf=Lfint & Ld=Ldint,

  V  L=[H|T] & compound(H) & H=..[F|Lh] &
     fnd_g2(Lh,Lf,LfT,Ldint1,Ldint) &
     fnd_g2(T,LfT,Lfint,Ld,Ldint1)

  V  L=[H|T] & ¬ compound(H) & member(H,Ldint) &
     fnd_g2(T,Lf,Lfint,Ld,Ldint)

  V  L=[H|T] & ¬ compound(H) & ¬ member(H,Ldint) &
     Lf=[H|Lf1] & Ldint1=[H|Ldint] &
     fnd_g2(T,Lf1,Lfint,Ld,Ldint1).
```

fnd_g2/5 est une généralisation de fnd_g/3:

```
fnd_g1(L,Lf,Ld) <=> fnd_g2(L,Lf,[],Ld,[]).
```

On remarque que deux appels récursifs sont nécessaires lorsque le premier élément de L est une structure. Une autre généralisation de fnd_g1/3 permet de n'avoir qu'un seul appel récursif.

procédure fnd_g3(L,Lf,Ld).

Soit
L une liste de listes,
Lf,Ld des listes.

Cette procédure détermine si

L=[L1,..,Ln], L' est la concaténation de L1,...,Ln. Lf est la liste des arguments de fnd(L') sans les termes qui apparaissent dans Ld.

```
fnd_g3(L,Lf,Ld) <=>
     L=[] & Lf=[]

  V  L=[H|T] & H=[] & fnd_g3(T,Lf,Ld)

  V  L=[H|T] & H=[H1|T1] & compound(H1) & H1=..[F|L1] &
     Lnew=[L1|T] & fnd_g3(Lnew,Lf,Ld).

  V  L=[H|T] & H=[H1|T1] & ¬ compound(H1) & member(H1,Ld) &
     Lnew=[T1|T] & fnd_g3(Lnew,Lf,Ld).

  V  L=[H|T] & H=[H1|T1] & ¬ compound(H1) & ¬ member(H1,Ld) &
     Lnew=[T1|T] & Lf=[H1|Lf1] & Ld1=[H1|Ld] &
     fnd_g3(Lnew,Lf1,Ld).
```

On remarque qu'un seul appel récursif est nécessaire dans tous les cas, il est néanmoins nécessaire de gérer des listes de listes.

5.3.3 Complexité des algorithmes.

Dans la méthodologie présentée au chapitre 1, on ne parle jamais de l'efficacité d'un algorithme qu'en termes d'efficacité de la procédure prolog dérivée. La propriété la plus importante d'un algorithme est alors de pouvoir en dériver une procédure avec la récursivité terminale.

La difficulté, si l'on veut parler de l'efficacité d'un algorithme logique, est qu'un algorithme logique est une formule bien formée en logique du premier ordre et que de ce fait cela n'a pas sens de parler d'exécution d'un algorithme logique. Cette difficulté peut être contournée car toutes les procédures logiques qui peuvent être dérivées d'un algorithme logique ont une propriété commune quelle que soit la manière dont elles sont exécutées.

Soit AL(P) un algorithme logique, P le programme obtenu en transformant AL(P) sous forme clausale et Q une question, **le nombre de branches succès d'un arbre de résolution induit par P U {Q} et la longueur de ces branches ne dépend pas de la règle de calcul**, [Naish 85] [Lloyd 87].

Le choix d'une bonne règle de calcul permet de diminuer le nombre de branches qui échouent ainsi que leur longueur.

Il est donc possible de parler de l'efficacité d'un algorithme en termes de complexité théorique des procédures qui peuvent en être dérivées. Par abus de langage on parlera de la complexité théorique de l'algorithme logique.

La complexité théorique est une notion bien connue en programmation, en voici une définition :

Lorsqu'on examine un algorithme A, on trouve presque toujours un paramètre, soit p, caractérisant la taille des données auxquelles A s'applique dans chaque cas. Soit T(p) le temps

d'exécution de A en fonction de p. On dit que **l'algorithme A est de complexité théorique O (f(p))**, où f est une certaine fonction du paramètre p, si la croissance asymptotique de T(p) est de l'ordre de f(p) au plus, ou autrement dit, si le rapport T(p)/f(p) est borné lorsque p tend vers + ∞. [Fichefet 84].

Par temps d'exécution, nous considérerons la longueur moyenne des branches succès.

5.3.3.1 exemple.

Le calcul de la complexité théorique d'un algorithme va être illustré sur deux algorithmes pour la procédure reverse/2 :

procédure reverse(L,L1).

Soit
L,L1 des listes.

Cette procédure détermine si

L1 est la liste L inversée.

Le premier algorithme est obtenu par induction sur L. Par facilité, nous le donnerons directement sous forme clausale, ainsi que l'algorithme de la procédure append/3 qui est utilisée pour construire l'algorithme.

```
reverse(L,L1) <- L=[] & L1=[].
reverse(L,L1) <- L=[H|T] & reverse(T,T1) & append(T1,[H],L1).

append(L,L1,L2) <- L=[] & L2=L1.
append(L,L1,L2) <- L=[H|T] & L2=[H|T2] & append(T,L1,T2).
```

Déterminons d'abord la complexité de append/3 en fonction de n la longueur de L :

-si n=0 alors T_app(n)=2,
-si n=1 alors T_app(1)= 2 + T_app(0) =4,
- ....
-si n=i alors T_app(i)= 2 + T_app(i-1) = 2 +... +2 = 2 (i+1).

La complexité de append/3 est donc O(n).

Déterminons maintenant le temps de reverse/2 en fonction de n la longueur de L :

-si n=0 alors T_rev(n)=2,
- ....
-si n=i alors T_rev(i)= 2 + T_rev(i-1) + T_app(i-1) = 2 (i-1+ i-2+ ... + 1) = i (i-1)

Dans le cas où n=i, il est possible d'additionner le temps de reverse/2 et de append/3 car comme T_rev(n) ne dépend pas de la règle de calcul, on peut en choisir une qui est linéaire, c-à-d qui termine complètement l'exécution d'un litéral avant de commencer l'exécution du suivant.

La complexité de reverse/2 est donc $O(n^2)$.

Au chapitre 1, j'ai construit un algorithme pour une version généralisée de reverse/2. je vais voir quelle est sa complexité théorique. Par facilité je donne directement l'algorithme sous forme clausale :

```
reverse_gen(L,L1) <- reverse(L,[],L1).

reverse(L,Lint,L1) <- L=[] & L1=Lint.
reverse(L,Lint,L1) <- L=[H|T] & Lint1=[H|Lint] &
                      reverse(T,Lint1,L1.
```

La complexité de reverse/2 est la même que la complexité de reverse/3. On va la calculer en fonction de n la longueur de L :

-si n=0 alors T(0)=2.
-....
-si n=i alors T(i)=2+T(i-1) = 2+ 2+....+2 +2 = 2(i+1).

La complexité de reverse_gen/2 est donc O(n).

On remarque que généraliser reverse/2 est non seulement intéressant en fonction de l'algorithme prolog dérivé, (il est possible d'obtenir une procédure avec la récursivité terminale pour reverse/3 et pas pour reverse/2) mais également en termes de complexité des algorithmes obtenus.

5.3.3.2 conclusion.

On vient de voir, comment mesurer la complexité théorique d'un algorithme. Il ne faut cependant pas perdre de vue que la procédure logique dérivée sera exécutée dans un langage donné (dans notre cas, Prolog) avec une règle de calcul déterminée. Il n'est donc pas inutile de regarder si la procédure dérivée est efficace, par exemple si on peut obtenir la récursivité terminale, etc... . Il ne faut pas non plus perdre de vue que pour atteindre les branches succès il est parfois nécessaire de contruire un certain nombre de branches qui échouent de longueur plus ou moins longue. C'est principalement pour cette raison que des algorithmes "generate and test" sont si inefficaces et qu'il est parfois nécessaire de construire une version récursive même si elle est de même complexité, ou de redéfinir la règle de calcul comme expliqué au chapitre deux. Par exemple le tri par permutation est de complexité $O(n^{n})$ selon notre mesure mais est moins efficace que le tri par insertion qui est de même complexité.

5.4 Procédures logiques.

Après avoir construit l'algorithme logique, il reste à dériver une procédure logique correcte puis à la transformer pour l'optimiser.

5.4.1 Correction.

Quand on dérive une procédure logique à partir d'un algorithme logique, cette procédure est déjà correcte par rapport à certains critères. D'autres doivent encore être vérifiés.

a. Négation

Pour qu'une procédure logique soit correcte, lors de la sélection d'un litéral négatif tous ses arguments doivent être des termes de base.

Quand dans un algorithme, il existe des litéraux $\neg \exists$ x1,...,xn: p(x1,...,xn,y1,...ym), la correction de la procédure est obtenue de manière un peu particulière.

Illustrons cela sur une version simplifiée de la procédure trans_pred/6.

Rappel:

Soit P un prédicat, Sg un ensemble d'atomes_2, Fp **le prédicat P transformé en fonction de Sg, noté Fp=tr_pred(P,Sg)**, est défini comme suit:
-s'il existe Ag ∈ Sg tel que Ag a même symbole de prédicat que P alors si P et Ag s'unifient avec mgu θ, P'=θ:var(P), Fp=ndv(flat(P',Ag),flat(Ag)),
-s'il n'existe pas Ag ∈ Sg tel que Ag a même symbole de prédicat que P alors Fp=P.

procédure trans_pred(P,Sg,Pnew)

Soit
P,Pnew des prédicats,
Sd une liste d'atomes_2.

Cette procédure détermine si

Pnew= tr_pred(P,Sg).

Directionnalité.

In(ngv,gr,var) Out(nc,gr,ngv). <1,1>.

Sans induction on obtient l'algorithme suivant :

```
trans_pred(P,Sg,Pnew) <=>
     functor(P,F,N) & member(Ag,Sg) & functor(Ag,F,N) &
     fl_ndv(P,Ag,Pnew)

  V  functor(P,F,N) &
     ¬ ∃ Ag: (member(Ag,Sg) & functor(Ag,F,N)) & Pnew=P.
```

à partir duquel on obtient la procédure logique :

```
(1)trans_pred(P,Sg,Pnew) ->
      functor(P,F,N),
      member(Ag,Sg),
      functor(Ag,F,N) &
      fl_ndv(P,Ag,Pnew).
(2)trans_pred(P,Sg,Pnew) ->
      functor(P,F,N)
      not(aux(Sg,F,N)) ,
      Pnew=P.

aux(Sg,F,N) :-
      member(Ag,Sg),
      functor(Ag,F,N).
```

Pour obtenir la deuxième clause, un nouveau prédicat aux/3 a été utilisé, de cette manière tous les paramètres de la négation Sg, F et N sont de base lors de sa sélection.

Comme (member(Ag,Sg) & functor(Ag,F,N)) et aux(Sg,F,N) sont incompatibles et que aux(Sg,F,N) ne calcule pas de résultat partiel, il est possible de supprimer la négation dans la seconde clause en plaçant un cut après le litéral functor(Ag,F,N) dans la première clause, (transformation 2). La procédure aux/3 peut alors également être supprimée.

Dans d'autres cas il est possible de supprimer la procédure aux/3, par une transformation basée sur une évaluation partielle (transformation 6). La clause (2) devient:

```
trans_pred(P,Sg,Pnew) ->
      functor(P,F,N)
      not(member(Ag,Sg),functor(Ag,F,N)) ,
      Pnew=P.
```

où malgré les apparences, les conditions sur la négation sont bien remplies au moment de la sélection de la négation.

b. Directionnalité.

La vérification de la directionnalité a un double but :

-la directionnalité est une précondition à l'utilisation d'une procédure. Lors de la sélection d'un litéral par la règle de calcul, il faut vérifier si cette précondition est bien respectée. Comme je l'ai indiqué plus haut, je m'assure habituellement que ces conditions peuvent être respectées lors de la construction de l'algorithme logique.
-étant donné la directionnalité en entrée, vérifier si la directionnalité en sortie est bien respectée. Si la condition précédente est respectée et que la directionnalité en sortie n'est pas respectée, il s'agit souvent d'une erreur dans la spécification qui est alors modifiée.

Dans le programme de génération de mode, la plupart des procédures sont prévues pour fonctionner pour une directionnalité bien précise. Cela facilite la construction du programme car il est plus facile de trouver un paramètre suffisamment instancié dans tous les usages possibles.

c. Types et autres préconditions.

Les types et autres préconditions sont des préconditions à l'utilisation d'une procédure. Cela peut être vérifié lors de la construction de l'algorithme logique. Ils constituent également une postcondition qu'il est nécessaire de vérifier pour que la procédure logique dérivée soit correcte.

d. Complétude et terminaison.

Lorsque la longueur de la séquence de substitutions réponses est finie, il est nécessaire de vérifier que la procédure se termine. Dans ce cas, la complétude est automatiquement obtenue.

Pour toutes les procédures écrites, la séquence de substitutions réponses était finie. La terminaison était chaque fois facile à démontrer, soit qu'il n'y avait pas de récursivité et que tous les arbres de résolution des sous-problèmes étaient finis, soit qu'il y avait récursivité mais que la procédure avait été construite avec le paramètre d'induction suffisamment instancié.

5.4.2 Transformations et optimisations.

Dans la méthodologie un certain nombre de transformations pour optimiser une procédure logique sont proposées. Ces transformations sont assez simples prises séparément, elles peuvent cependant interférer les unes sur les autres. Le plus simple est souvent d'avoir une idée sur les optimisations que l'on veut réaliser, et de vérifier qu'il est possible d'obtenir la procédure prolog désirée en appliquant une séquence de transformations à la procédure logique de départ.

La transformation de la procédure f_ndv_1/5 présentée au chapitre 3 montre bien comment plusieurs transformations sont utilisées successivement pour obtenir la procédure prolog définitive.

D'une manière générale on peut remarquer que ces transformations réduisent la lisibilité du programme. Ce n'est toutefois pas très grave puisque l'on dispose également des spécifications et des algorithmes logiques des différentes procédures du programme.

Généralement, je n'ai pas essayé d'optimiser au maximum mes procédures. Dans la suite, je reprendrai uniquement les transformations que j'ai le plus souvent appliquées .

a. Récursion terminale.

Vérifier dès la construction de la procédure logique, qu'il est possible de dériver une clause correcte permet également de vérifier qu'il est possible de placer l'appel récursif en dernière position. Quand cela n'était pas possible, j'ai systématiquement généralisé la procédure.

Pour obtenir la récursion terminale, il faut également que la séquence de litéraux précédant l'appel récursif soit déterministe. Il est souvent nécessaire d'insérer un cut pour permettre au système de déterminer que ces litéraux sont déterministes. Normalement l'appel récursif doit se trouver dans la dernière clause. Quand ce n'est pas le cas, un cut peut permettre d'indiquer qu'aucune autre clause de la procédure ne peut être utilisée comme alternative, cela est équivalent à avoir l'appel récursif dans la dernière clause.

La récursivité terminale permet de gagner en temps d'exécution et en place mémoire.

b. Tranformations basées sur une évaluation partielle.

Quand une procédure a été généralisée pour être construite, cette procédure a souvent la forme d'une clause p(x1,...,xn) :- p_gen(x1,...,xn,y1,...,ym). La transformation 6 permet d'appeler directement la procédure généralisée.

Représenter des données par types abstraits a souvent comme effet d'augmenter le nombre d'appels de procédure au niveau exécution du programme. La transformation 6 permet dans certains cas d'éliminer ces appels de procédure.

exemple:

Si la procédure empty_sub/1 est définie par la clause

```
empty_sub([]).
```

alors la clause

```
trans_cl(Cl,Sg,Clnew,Mode) <-
     Cl=(H:-T),
     Clnew=(Htrans:-Ttrans),
     empty_sub(Es),
     trans_pred(H,Sg,Htrans,Es,Subst,Mode),
     trans_tail(T,Sg,Ttrans,Subst).
```

peut être transformée en

```
trans_cl(Cl,Sg,Clnew,Mode) <-
     Cl=(H:-T),
     Clnew=(Htrans:-Ttrans),
     Es =[],
     trans_pred(H,Sg,Htrans,Es,Subst,Mode),
     trans_tail(T,Sg,Ttrans,Subst).
```

ce qui évite un appel de procédure et permet également d'autres transformations.

c. Transformations basées sur l'égalité.

Les transformations basées sur l'égalité permettent de ramener certaines égalités dans la tête de la clause, ce qui les fait ressembler plus aux procédures que l'on trouve habituellement dans les livres prolog. Elles permettent également de propager les égalités vers la fin de la clause et ainsi d'éventuellement en supprimer certaines.

exemple:

si je reprends la clause de l'exemple précédent et que je lui applique les transformations 7 à 10, j'obtiens la clause :

```
trans_cl((H:-T),Sg,(Htrans:-Ttrans),Mode) <-
     trans_pred(H,Sg,Htrans,[],Subst,Mode),
     trans_tail(T,Sg,Ttrans,Subst).
```

d. Transformations basées sur des arbres de résolution équivalents.

En évitant de construire plusieurs sous-arbres de recherche identiques, ces tranformations permettent d'améliorer l'exécution du programme. Dans certains cas les cut introduits peuvent également servir pour indiquer la récursivité terminale.

Le gain obtenu par la transformation 1 peut être réalisé sans introduire de nouvelles procédures en utilisant l'opérateur ";" (qui signifie "ou")

rappel :

```
p(x1,...,xn) <- T, S1.
p(x1,...,xn) <- T, S2.
```

est transformé en

```
p(x1,...,xn) <- T, p1(y1,...,ym).
p1(y1,...,ym) <- S1.
p1(y1,...,ym) <- S2.
```

où -y1,...,Ym est l'ensemble des variables de S1 et S2.
-p n'a pas d'effet secondaire.
-T,S1,S2 ne contiennent pas de cut,
-p n'est pas infini.

il est également possible de transformer cette procédure en :

```
p(x1,...,xn) <- T, (S1;S2).
```

Cette transformation peut également être utilisée quand la séquence de litéraux identiques ne se trouve pas au début de la queue de la clause.

```
p(x1,...,xn) <- S1, T, S2.
p(x1,...,xn) <- S3, T, S4.
-------------------------------------
p(x1,...,xn) <- (S1;S3), T, (S2;S4).
```

la taille des arbres de recherche peut également être réduite en plaçant les litéraux qui peuvent échouer le plus près possible de la tête de la clause.

5.5 Types abstraits.

Avant de terminer ce chapitre, je vais encore faire quelques remarques sur l'utilisation de types abstraits.

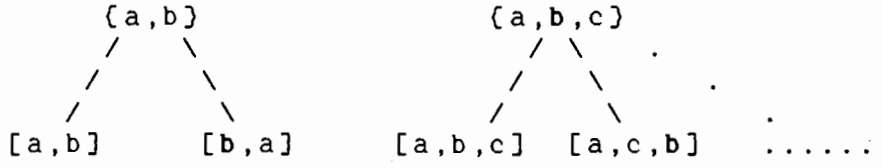
Dans la phase de spécification, les types abstraits permettent de se concentrer sur les propriétés d'un objet sans tenir compte de détails d'implémentation. Habituellement un objet est plus abstrait que les différents objets disponibles dans le langage. Il est alors nécessaire par **raffinement de données** de choisir une **représentation** pour l'objet abstrait. Il est également nécessaire que les opérations sur la représentation **modélisent** les opérations sur l'objet abstrait. Une représentation est reliée à l'objet dont elle est le raffinement par une **fonction de récupération** (retreive function).

Représenter un objet par types abstraits a l'avantage de permettre de choisir la représentation de données après la phase de spécification. A ce moment on connait les différentes opérations qui vont devoir être réalisées, ce qui permet de choisir la représentation la plus adéquate. La représentation choisie peut également être cachée dans un module, ce qui permet de modifier cette représentation de manière transparente pour le reste du programme.

Souvent le choix de représentation entraîne que plusieurs valeurs de la représentation corrrespondent à une valeur abstraite. Il y a une relation 1-N entre l'objet et sa représentation. [Jones 80].

exemple:

Un ensemble peut être représenté par la liste des éléments de cet ensemble, plusieurs listes corespondent au même ensemble.



Soit f la fonction de récupération f([a,b])=f([b,a])={a,b} et f([a,b,c])=f([a,c,b])={a,b,c}.

Quand, comme dans l'exemple, plusieurs valeurs de la représentation corespondent à la même valeur de l'objet abstrait alors la procédure d'unification "=" ne permet pas de modéliser l'égalité sur l'objet abstrait.

exemple :

[a,b] = [b,a] ou [a,b,c] = [a,c,b] sont faux alors que {a,b}={a,b} et {a,b,c}={a,b,c}.

En prolog, ce problème peut exister de manière générale. Par exemple si on pose la question:

?- X is 4 / 2, Y is 2, X=Y, (avec / la divison sur les réels),

certains systèmes prolog répondent no, car X est 2 représenté en virgule flottante et Y est représenté en décimale.

Il faut également remarquer que ce problème ne se pose pas si une des deux représentations est une variable.

Ce problème peut être solutionné en imposant qu'à une valeur de l'objet abstrait corresponde une seule valeur de la représentation. Dans l'exemple cela peut être fait en triant les éléments de la liste. Cette solution a le désavantage de demander des calculs supplémentaires pour obtenir la représentation d'une valeur d'un objet.

Une autre solution est de construire une procédure particulière pour comparer les objets abstraits. Par rapport à la solution précédente celle-ci nécessite des calculs lors de la comparaison de deux représentations. L'une ou l'autre solution sera choisie suivant qu'il est nécessaire ou non de comparer régulièrement des objets.

Quand on écrit l'algorithme logique, on fait des opérations sur les données abstraites, il est donc préférable, même si on a choisi la seconde solution, d'utiliser la procédure "=". Une procédure particulière est alors utilisée lors de la dérivation de la procédure logique quand les deux arguments de "=" ne sont pas des variables.

Dans le programme de génération de déclarations de mode un des deux arguments de "=" était toujours une variable, il n'y a donc jamais eu de problème.

Savoir qu'un argument est toujours une variable et donc qu'il n'est pas nécessaire de se soucier des différentes représentations possibles permet également de simplifier la construction de certaines procédures.

Illustrons cela avec la procédure L_to_Subs qui permet de transformer une représentation externe d'une substitution en une représentation interne :

procédure L_to_subs(L,Subs)

Soit
L une liste de liens (V/term).
Subs une substitution.

pre: si L=[V1/t1,....,Vn/tn] alors **V** i, ti est différent de Vi et V1,...,Vn sont toutes des variables différentes.

Cette procédure détermine si

L est la liste des éléments de Subs.

Directionnalité.

In(L(ngv),Var) Out(nc,S(ngv)). <1,1>.

Dans le programme, une substitution est représentée par une liste de liens (V/Term). Pour faciliter la composition de deux substitutions, la liste peut contenir plusieurs liens pour la même variable. Dans ce cas, seul le premier lien pour cette variable est un lien de la substitution.

Etant donné notre choix de représentation, la procédure logique pour la directionnalité désirée peut être facilement écrite :

L_to_Subs(L,Subs) :- Subs=L.

Si on écrit un algorithme logique, celui-ci doit également être correct pour, par exemple, obtenir la liste des liens d'une substitution ou pour vérifier qu'une liste de liens est identique à la liste des liens d'une substitution. On est alors obligé de tenir compte que plusieurs valeurs de la représentation peuvent correspondre à la même substitution. Construire un algorithme logique et en dériver une procédure logique est alors beaucoup plus compliqué.

5.6 Conclusion.

D'une manière générale, la méthodologie est appropriée pour construire un programme tel que le programme de génération de mode. De légères modifications ont bien été apportées lors de son utilisation mais celles-ci ne changent pas l'esprit de la méthodologie.

Je pense que l'important est de bien découper le programme en procédures dont la spécification peut être exprimée de manière simple. Cette première phase de spécification ne doit pas nécessairement se faire de manière Top-Down comme la construction des algorithmes. N'importe quelle méthode de génie logiciel est applicable. La méthodologie est alors utilisée pour contruire chaque procédure séparément. Lors de la programmation, de nouvelles procédures sont éventuellement spécifiées et construites, par exemple lorsqu'on généralise une procédure.

Si dans le programme, il existe un certain nombre de procédures pour lesquelles la méthodologie est moins appropriée (interface utilisateur, extra logique, etc ...), celles-ci peuvent être construites avec une autre méthodologie.

Il y a beaucoup de chances pour que, si on a choisi Prolog comme langage de programmation, la spécification de la plupart des procédures soit principalement la description d'une relation entre les arguments de la procédure. La méthodologie est alors appropriée pour résoudre ce problème.

Conclusion

Le but du mémoire était d'appliquer et de tester une méthodologie de programmation logique [Deville 87].

Cette méthodologie couvre les différentes phases de construction, de la spécification jusqu'à l'obtention d'un programme prolog correct et efficace.

Pour tester la méthodologie, j'ai écrit un programme d'application. Ce programme permet de transformer des programmes prolog pour générer des déclarations de mode plus précises.

Actuellement le programme permet de transformer des programmes obtenus par compilation d'informations de contrôle. Il pourrait être simplifié si l'on disposait de l'arbre de résolution symbolique du programme compilé à la place de l'arbre de résolution symbolique du programme de départ. Les principes de transformation, bien que principalement intéressants pour les programmes obtenus par compilation d'informations de contrôle sont également applicables à d'autres programmes.

Dans certains cas, les transformations réalisées peuvent entraîner une perte de performance plutôt qu'un gain. Le programme devrait être amélioré afin de détecter les cas défavorables.

Si l'on considère que le cycle de vie d'un programme est composé de cinq étapes : analyse fonctionnelle, design, implémentation, tests et maintenances, la méthodologie proposée est principalement concernée par la phase d'implémentation.

Avant d'utiliser la méthodologie, le programme a d'abord été découpé en niveaux et en modules et la plupart des données ont été représentées par types abstraits. Cette découpe permet de cacher dans des modules les parties du programme qui ont un caractère purement déclaratif.

Une fois la phase de design terminée, la méthodologie a été utilisée pour construire différentes parties du programme. L'utilisation de la méthodologie s'est avérée très profitable.

Excepté pour une procédure et quelques erreurs de syntaxe, toutes les procédures construites se sont révélées correctes par rapport à leur spécification.

Lors de la construction du programme, certaines modifications ou extensions ont été apportées à la méthodologie.

Dans la méthodologie, la construction d'une procédure est présentée en deux phases. Dans une première phase, on construit un algorithme logique correct d'un point de vue déclaratif. Dans une seconde phase, une procédure logique est dérivée, sa correction est vérifiée d'un point de vue procédural, puis elle est optimisée. On remarque que les procédures construites doivent être correctes d'un point de vue déclaratif et procédural.

Cette découpe permet d'expliquer facilement la méthodologie. Dans la documentation du programme, il est également intéressant d'avoir l'algorithme logique plus clair que la procédure prolog optimisée.

Lors de la construction d'une procédure, la distinction entre les deux phases n'est pas aussi marquée. Lors de la construction de l'algorithme logique, je vérifie déjà qu'il est possible de dériver une procédure logique correcte et efficace. Cela permet en étant un peu plus permissif sur la correction d'un algorithme logique de supprimer un certain nombre de litéraux de vérification des préconditions.

On a indiqué, dans la présentation de la méthodologie, que la construction d'une procédure logique est simplifiée si le paramètre d'induction est un terme de base. Ce résultat peut être étendu à des paramètres suffisamment instanciés. Les notations pour décrire la directionnalité d'une procédure doivent être étendues pour pouvoir indiquer qu'un paramètre est suffisamment instancié.

Dans la méthodologie, l' efficacité d'un algorithme logique est présentée en termes d'efficacité de la procédure logique dérivée. Etant donné que le nombre et la longueur des substitutions réponses ne dépendent pas de la règle de calcul, j'ai proposé une façon de calculer l'efficacité d'un algorithme logique en termes de complexité théorique.

Lors de la phase de design du programme, j'ai décidé de représenter certaines données par types abstraits. Certaines précautions doivent être prises lorsqu'on utilise des types abstraits en programmation logique, principalement avec l'unification. Deux solutions ont été présentées à ce problème.

D'une manière générale, la méthodologie était appropriée pour construire l'application réalisée car les modifications ou extensions qui ont dû y être apportées restent dans l'esprit de celle-ci.

Bibliographie

[BIM 86]
BIM Prolog Manual. BIM S.A., Everberg Belgium 86.

[Bruynooghe 82]
Bruynooghe M.: The Memory Management of Prolog Implementation. in Logic Programming, K.L. Clark and S. Tarnlund, Academic Press, London, 1982.

[Bruynooghe 86]
Bruynooghe M., De Schreye D., Krekels B.: Compiling Control. Report CW 47, K.U.L. Departement of Computer Science, March 86.

[Burstall 77]
Burstall R.M., Darlington J.: A Transformation System for Developing Recursive Program. J.ACM, Vol 24(1), January 77, pp 44-67.

[Clocksin 81]
Clocksin W.F., Mellish C.S.: Programming in Prolog. Springer Verlag, Berlin 1981.

[De Schreye 87]
De Schreye D. Bruynooghe M.: On the Transformation Of Logic Programs with Instantiation Based Computation Rules. Report CW 55, K.U.L. Departement of Computer Science, May 87.

[Debray 86a]
Debray S.K., Warren D.S.: Detection and Optimisation of Functionnal Computations in Prolog. Proc. Third Int. Conf. on Logic Programming, London, July 86.

[Debray 86b]
Debray S.K., Warren D.S.: Automating Mode Inferencing for Prolog Programs. Proc. 1986 Logic Programming Symposium, Salt Lake City, IEEE Society Press, Sept 86, pp 78-88.

[Deville 87]
Deville Y.: A Methodology for Logic Program Construction. Ph.D. Thesis, University of Namur, Belgium February 87.

[Fichefet 85]
Fichefet J.:Théorie des Graphes : Syllabus. Institut d'informatique, F.N.D.P. Namur, 85.

[Jones 80]
Jones C.B.: Software Development: A Rigourous Approach. Prentice Hall International 80, pp 179-193.

[Kowalski 74]
Kowalski R.A.: Predicate Logic as a Programming Language. IFIP 74, pp 569-574.

[Kowalski 79]
Kowalski R.A.: Algorithm = Logic + Control. C.ACM 22(7), July 79, pp 424-436.

[Lloyd 87]
Lloyd J.W.: Foundations of Logic Programming: Second Extended Edition. Springer Verlag 87.

[Mellish 81]
Mellish C.S.: The Automatic Generation of Mode Declarations for Prolog Programs. DAI Researsh Paper 163, Dept. of Artificial Intelligence, University of Edinburgh, August 81.

[Naish 85a]
Naish L.: Prolog Control Rules. Proc IJCAI, Los Angeles 1985, pp 720-723

[Naish 85b]
Naish L.: Automating Control for Logic Programs. Journal of Logic Programming, 1985(3), pp 73-77.

[Sterling 86]
Sterlind L., Shapiro E.: The Art of Prolog: Advanced Programming Techniques. The MIT Press, 1986.

Table des matières

Texte du programme

Cette annexe contient la spécification et les algorithmes de toutes les procédures du programme réalisé. Ces procédures sont présentées module par module suivant la découpe proposée au chapitre 3. Dans chaque module sera d'abord donnée la spécification des procédures qui constituent l'interface du module, ensuite si nécessaire la représentation des données choisie pour implémenter le module, finalement la spécification des procédures internes au module et les algorithmes logiques et les procédures prolog de toutes les procédures du module.

1. Module termes prolog, termes_0, 1 et 2.

Pour des facilités de lecture, les procédures de ce module seront séparées en deux groupes, celles relatives aux termes prolog et celles relatives aux termes_0 et termes_2.

1.1 Spécification de l'interface.

a. Termes prolog.

procédure var(X).

Soit
X un terme.

Cette procédure détermine si

X est une variable.(E L).

Directionnalité.

```
In(var)        Out(var). <1,1>.
In(novar)                <0,0>.
```

****

procédure compound(T).

Soit
T un terme.

Cette procédure détermine si

T est f(t1,...,tn) avec f un foncteur quelconque et n un entier strictement positif. (E.L).

Directionnalité.

```
In(var)                    <0,0>.
In(novar)      Out(novar).<0,1>.
```

****

procédure atomic(X)

Soit
X un terme

Cette procédure détermine si

X est un atome ou un entier (E.L)

Directionnalité.

```
In(gr)    Out(gr). <0,1>.
In(var)              <0,0>.
```

****

procédure functor(T,F,N).

Soit
T un terme.

Cette procédure détermine si

∃ un foncteur F et un entier n tels que T=F(t1,...,tn).

N.B un atome est considéré comme une structure d'arité 0.

Directionnalité.

```
In(nonvar,any,any)       Out(nonvar,gr,gr). <0,1>.
In(var,gr,gr)            Out(ngv,gr,gr).    <1,1>.
```

****

procédure =..(T,L).

Soit
T un terme,
L une liste.

Cette procédure détermine si

∃ un foncteur f et un entier n tels que T=f(t1,...,tn) alors L est la liste [f,t est la liste [f,t1,...,tn].

N.B. =.. est défini comme un opérateur, et est habituellement appelé "univ".

Directionnalité.

In(nonvar,any) Out(nonvar,L(any)). <0,1>.
In(any,L(any)) Out(nonvar,L(any)). <0,1>.

****

procédure ==(T,T1).

Soit
T, T deux termes.

Cette procédure détermine si

T et T1 sont syntaxiquement identiques.

N.B. == est défini comme un opérateur.

Directionnalité.

In(any,any) Out(any,any). <0,1>.

****

b. Termes_0 et Termes_1.

procédure is_Vi(T).

Soit
T un terme_2.

Cette procédure détermine si

T est une variable_2 Vi, (i∈N).

Directionnalité.

In(gr) Out(gr). <0,1>.

****

procédure is_Ni(T).

Soit
T un terme_2.

Cette procédure détermine si

T est une variable_2 Ni, (i∈N).

Directionnalité.

In(gr) Out(gr). <0,1>.
In(var) Out(gr). <1,1>.

****

procédure is_Gi(T).

Soit
T un terme_2.

Cette procédure détermine si

T est une variable_2 Gi, (i∈N).

Directionnalité.

In(gr) Out(gr). <0,1>.
In(var) Out(gr). <1,1>.

****

procédure is_const(T).

Soit
T un terme_2.

Cette procédure détermine si

T est une constante.

Directionnalité.

In(gr) Out(gr). <0,1>.

****

procédure f_vi(X,Y,Z)

Soit
X,Y,Z des variables_2 Vi, (i∈N).

Cette procédure

définit une fonction injective fct:N^{∞}->N telle que si X=Vi et Y=Vj alors Z=Vk avec i,j,k ∈ N et k=fct(i,j).

effets de bord :

Des faits v_fvi peuvent être ajoutés à la B.D. et la valeur de Last_Val(N). modifiée par cette procédure.

Directionnalité.

In(gr,gr,var) Out(gr,gr,gr). <1,1>.

****

procédure is_v(T).

Soit
T un terme_0.

Cette procédure détermine si

T est v.

Directionnalité.

In(gr) Out(gr). <0,1>.

****

procédure is_g(T).

Soit
T un terme_0.

Cette procédure détermine si

T est g.

Directionnalité.

In(gr) Out(gr). <0,1>.

****

procédure is_any(T).

Soit
T un terme_0.

Cette procédure détermine si

T est any.

Directionnalité.

In(gr) Out(gr). <0,1>.

****

procédure fo(T1,T0),

Soit
T1 un terme_1,
T0 un terme_0.

Cette procédure détermine si

T0=Fo(T1).

Directionnalité.

In(gr,gr) Out(gr,gr). <0,1>.

****

1.2 Choix de représentation.

Un atome_0 est représenté par un terme prolog comme suit:
-le terme_0, T0 est représenté par un atome = T0,
-l'atome_0, P(t1,...,tn) par le terme P(t1,...,tn).

Un atome_2 est représenté par un terme prolog comme suit:
-une constante est représentée par cette même constante.
-une variable Vi par la constante vi, (i ∈ N),
-une variable Gi par la constante g,(i ∈ N)
-une variable Ni par la constante any,(i ∈ N)

remarque: cette représentation impose que les atomes_2 à représenter ne contiennent pas de constantes any, g, ou vi (i ∈ N).

1.3 Spécification des procédures internes.

procédure int_to_list(Int,L).

Soit
Int un entier positif,
L une liste non vide d'entiers compris entre 48 et 57.

Cette procédure détermine si

L est la liste des codes ASCII de la représentation décimale de Int.

Directionnalité.

In(gr,var) Out(gr,gr). <1,1>.

****

procédure int_to_list(Int,Lt,L).

Soit
Int un entier positif,
L deux listes non vides d'entiers compris entre 48 et 57.

Cette procédure détermine si

L est la concaténation de la liste des codes ASCII de la représentation décimale de Int et de Lt.

Directionnalité.

In(gr,gr,var) Out(gr,gr,gr). <1,1>.

****

procédure list_int(L).

Soit
L une liste d'entiers positifs.

Cette procédure détermine si

L est une liste non vide dont les éléments sont compris entre 48 et 57.

Directionnalité.

In(gr) Out(gr). <0,1>.

****

procédure fo_L(L1,L0).

Soit
L1 une liste d'atomes_1,
L0 une liste d'atomes_0.

Cette procédure détermine si

L1=[t1,...,tn] et L0=[k1,...,kn] et ∀ i, 1≤i≤n, Ki=Fo(ti).

Directionnalité.

In(gr,gr) Out(gr,gr). <0,1>.

****

procédure L_g(L).

Soit
L une liste d'atomes_2.

Cette procédure détermine si

L=[a1,...,an] et ∀ i fo(ai)=g.

Directionnalité.

In(gr) Out(gr). <0,1>.

****

1.4 Algorithmes et procédures logiques.

Les procédures var(X), functor(T,F,N), =..(T,L), ==(T,T1) sont des primitives du langage.

procédure: compound(T).

Procédure prolog:

```
compound(T) :-
          not(var(T)),
          not(atomic(T)).
```

****

procédure: is_vi(T).

Procédure prolog:

```
is_vi(X) :-
      atom(X),
      name(X,[118|I]),
      list_int(I).
```

****

procédure: is_Gi(T).

Procédure prolog:

```
is_gi(g).
```

****

procédure: is_Ni(T).

Procédure prolog:

```
is_Ni(any).
```

****

procédure: is_const(T).

Procédure prolog:

```
is_const(T) :-
          not(T=g),
          not(T=any),
          not(is_vi(T)),
          atomic(T).
```

****

procédure: f_vi(X,Y,Z).

Procédure prolog:

```
f_vi(X,Y,Z) :-
          v_fvi(X,Y,Z),
          is_vi(Z),
          !.
f_vi(X,Y,Z) :-
          last_val(Val),
          Val1 is Val+1,
          int_to_list(Val1,[]Lval1),
          name(Z,[118|Lval1]),
          assert(v_fvi(X,Y,Z),
          retract(last_val(Val)),
          assert(last_val(Val1)).

last_val(0).
```

****

procédure: is_v(T).

Procédure prolog:

```
is_v(v).
```

****

procédure: is_g(T).

Procédure prolog:

```
is_g(g).
```

****

procédure: is_any(T).

Procédure prolog:

```
is_any(any).
```

****

procédure: int_to_list(Int,L).

Procédure prolog:

```
int_to_list(Int,L) :-
          int_to_list(Int,[],L).
```

****

procédure: int_to_list(Int,Lt,L).

Algorithme logique:

```
int_to_list(Int,Lt,L) <=>
     Int <  10 & H is int +48 & L=[H|Lt],
   V Int >= 10 & E is Int mod 10 + 48 &
     Int1 is Int // 10 & int_to_list(Int1,[E|Lt],T).
```

Procédure prolog:

```
int_to_list(Int,Lt,L) :-
          Int >= 10,
          !,
          E is Int mod 10 + 48,
          Int1 is Int // 10,
          int_to_list(Int1,[E|Lt],T).
int_to_lis(Int,Lt,[H|Lt]) :-
          H is Int + 48.
```

****

procédure: list_int(L)..

Procédure prolog:

```
list_int([H|T]) :-
                  H>48,
                  H<58,
                List_int(T).
list_int([H]) :-
                H>48,
                H<58.
```

****

procédure: fo(T1,T0).

Procédure prolog:

```
fo(_,any).
fo(T1,X) :-
          is_vi(T1),
          X=v,
          !./
fo(T1,g) :-
          atomic(T1),
          !.
fo(T1,g) :-
          T1=..[F|L1],
          L_g(L1).

fo(T1,T0) :-
          not(atomic(T0)),
          T1=..[F|L1],
          T0=..[F|L0],
          fo_L(L1,L0],
          !.
```

****

procédure: L_g(L)

Procédure prolog:

```
L_g([]).
L_g([H|T]) :-
      fo(H,g),
      L_g(T).
```

****

procédure: fo_L(L1,L0).

Procédure prolog:

```
fo_L([H1|T1],[H0|T0]) :-
          fo(H1,H0),
          fo_L(T1,T0).
fo_L([],[]).
```

****

2 Module "listes".

Ce module offre deux types différents de listes , soit la liste prolog habituelle, soit la liste associative.

2.1 Spécification de l'interface.

a. liste.

procédure append(L,L1,Lres).

Soit
L,L1,Lres des listes.

Cette procédure détermine si

Lres est la concaténation de L et de L1.

Directionnalité.

```
In(L(any),var,var)          Out(nc,nc,ngv).<1,1>.
In(L(any),L(any),any)            Out(nc,nc,L(any)). <0,1>.
```

****

procédure member(E,L).

Soit
E un terme,
L une liste.

Cette procédure détermine si

E est un élément de L.

Directionnalité.

In(any,L(any)) Out(any,L(any)). <0,n>.

****

procédure member(E,L,L1)

Soit
E un terme,
L,L1 deux listes

Cette procédure détermine si

E est élément de L et L1 est L sans la première occurrence de E dans L.

Directionnalité.

In(any,L(any),var) Out(any,L(any),L(any)). <0,1>.
In(any,var,L(any)) Out(any,L(any),L(any)). <0,1>.

****

procédure add_S(S,L,S1).

Soit
S,L,S1 des listes,

Cette procédure détermine si

S1 est la liste S à laquelle ont été ajoutés les éléments de L qui n'y figurent pas encore.

Directionnalité.

In(gr,L(any),var) Out(gr,L(any),L(any)). <1,1>.

****

b. liste associative.

procédure f_elem(L,H,L1)

Soit
L,L1 deux listes associatives,
H un terme.

Cette procédure détermine si

H est une variante du premier élément de L. L1 est L sans ce premier élément.

Directionnalité.

In(L(any),any,var) Out(L(any),any,L(any)). <0,1>.

****

procédure add_al(L,E,L1).

Soit
L,L1, des listes associatives,
E un terme.

Cette procédure détermine si

L1 est la liste L à laquelle on a ajouté E en dernière position.

Directionnalité.

In(L(any),any,var) Out(nc,any,L(any)). <1,1>.

****

procédure empty_al(Name,L)

Soit
L une liste associative,
Name un atome.

Cette procédure détermine si

L est la liste vide de nom L.

effets de bord.

la liste L est créée pour tout le programme, sa portée ne se limite pas à la clause où elle a été créée.

Directionnalité.

In(gr,var) Out(gr,L(any)). <1,1>.
In(any,novar) Out(gr,nc). <0,1>.

****

2.2 Choix de représentation.

Une liste associative est un triple (Name,Inf,Sup). Name est le nom de la liste, Inf le numéro associé au premier élément de la liste et Sup le numéro associé au dernier élément de la liste.

Les éléments de la liste sont représentés par Sup-Inf+1 relation ass_1(N,Name,Elem) avec N un entier positif Inf≤N≤Sup, qui détermine la position de Elem dans la liste de nom Name.

2.3 Algorithmes et procédures logiques.

procédure: append(L,L1,Lres).

Algorithme logique:

```
append(L,L1,Lres) <=>
          L=[] & Lres=L1
       V  L=[H|T] & Lres=[H|Tres] & append(T,L1,Tres).
```

Procédure prolog:

```
append([H|T],L1,[H|Tres]) :-
          append(T,L1,Tres).
append([],Lres,Lres).
```

****

procédure: member(E,L).

Algorithme logique:

```
member(E,L) <=>
          L=[] & false
       V  L=[H|T] & E=H
       V  L=[H|T] &  E=/=H & member(E,T).
```

Procédure prolog:

```
member(E,[E|T]).
member(E,[H|T]). :-
          member(E,T).
```

****

procédure: member(E,L,L1).

Algorithme logique:

```
member(E,L,L1) <=>
          L=[] & false
       V  L=[H|T] & E=H & L1=T
       V  L=[H|T] & E=/=T & member(E,T,T1) & L1=[H|T1].
```

Procédure prolog:

```
member(E,[E|T],T) :-
          !.
member(E,[H|T],[H|T1]) :-
          member(E,T,T1).
```

****

procédure: add_S(S,,S1).

Algorithme logique:

```
add_S(S,L,S1) <=>
      L=[] & S1=S,
   V  L=[H|T] & member(H,S) & add_S(S,T,S1).
   V  L=[H|T] & ¬ member(H,S) & S1=[H|Ts1] &
      add_S(S,T,Ts1).
```

Procédure prolog:

```
add_S(S,[],S).
add_S(S,[H|T],S1) :-
          member(H,S),
          !
          add_S(S,T,S1).
add_S(S,[H],[H|Ts1]) :-
          add_S(Sint,T,Ts1).
```

****

procédure: add_al(L,E,L1).

Procédure prolog:

```
add_al((Name,Inf,Sup),E,(Name,Inf,Sup)) :-
      L=(Name,Inf,Sup)
      Sup1 is Sup +1,
       assert(ass_l(Sup1,Name,E).
```

****

procédure: empty_al(L).

Procédure prolog:

```
empty_al(Name,L) :-
      var(L),
      !,
      L=(Name(1,0)).
empty_al(Name,(Name,Inf,Sup)) :-
      Inf > Sup.
```

****

procédure: f_elem(L,H,L1).

Procédure prolog:

```
f_elem((Name,Inf,Sup),E,(Name,Inf1,Sup) :-
      Inf =< Sup,
      Inf1 is Inf+1,
      ass_l(Inf,Name,E).
```

****

3 Module "Substitution".

3.1 Spécification de l'interface.

procédure empty_Sub(S).

Soit
S une substitution

Cette procédure détermine si

S est la substitution vide.

Directionnalité.

In(any) Out(gr). <0,1>.

****

procédure L_to_subs(L,Subs)

Soit
L une liste de liens (V/term).
Subs une substitution.

pre: si L=[V1/t1,....,Vn/tn] alors ∀ i,ti est différent de Vi et V1,...,Vn sont toutes des variables différentes.

Cette procédure détermine si

L est la liste des éléments de Subs.

Directionnalité.

In(L(ngv),Var) Out(L(ngv),S(ngv)). <1,1>.

****

procédure sub_to_conj(Sub,Ctail,C).

Soit
Sub une substitution,
Ctail, C des conjonctions de termes.

Cette procédure détermine si

Sub={(V1/T1),...,(Vn/Tn)} alors C=(V1=T1),...,(Vn=Tn),Ctail.

Directionnalité.

In(S(ngv),any,var) Out(nc,nc,ngv). <1,1>.

****

procédure substitue(T,Subst,Ts).

Soit
T,Ts deux termes,
Subst une substitution.

Cette procédure détermine si

Ts est T Subst, l'instance de Ts selon Subst.

Directionnalité.

In(any,S(ngv),var) Out(nc,nc,gr). <1,1>.

****

procédure unify(T1,T2,T,M).

Soit
T1,T2,T des termes,
M une substitution.

Cette procédure détermine si

T1 s'unifie à T2 avec mgu M, et T est T1 M.

Directionnalité.

In(any,any,var,var) Out(nc,nc,any,S(ngv)). <0,1>.

****

procédure unifier(T1,T2,M).

Soit
T1,T2 deux termes,
M une substitution.

Cette procédure détermine si

T1 s'unifie à T2 avec mgu M.

Directionnalité.

In(any,any,var) Out(nc,nc,S(ngv)). <0,1>.

****

procédure comp(S1,S2,S).

Soit
S1,S2,S des substitutions.

Cette procédure détermine si

S est la composition de S1 et S2.

Directionnalité.

In(S(ngv),S(ngv),var) Out(nc;nc,S(ngv)). <1,1>.

****

procédure unifier_2(T,Tg,Subst).

Soit
T un terme,
Tg un terme_2,
Subst une substitution.

Cette procédure détermine si

θ est le mgu de T et Tg, et Subst=$\theta_{|var(T)}$.

Directionnalité.

In(any,gr,var) Out(nc,gr,S(ngv)). <0,1>.

****

3.2 Choix de représentation.

Une substitution sera représentée par une liste de termes de la forme (V/Term). La liste peut contenir plusieurs liens pour la même variable, dans ce cas seul le premier lien pour cette variable est un lien de la substitution. On n'impose pas non lus que V soit différent de Term.

3.4 Spécification des procédures internes.

procédure subs_L(L,Subs,Ls).

Soit
L,Ls deux listes,
Subs une substitution.

Cette procédure détermine si

$L=[t_1,...,t_n]$ et $Ls=[ts_1,...,ts_n]$ et $\forall$ i, $1 \leq i \leq n$ et ts_i est t_i Subs.

Directionnalité.

In(L(any),S(any),var) Out(nc,nc,L(any)). <1,1>.

****

procédure member_subs(V,Term,Subs).

Soit
V,Term des termes,
Subs une substitution.

Cette procédure détermine si

Subs contient un lien (V,Term).

Directionnalité.

In(var,any,S(ngv)) Out(nc,nc,nc). <0,1>.

****

procédure unifier_L2(L,Lg,Sint,Subst,Lvt,Lvi).

Soit
L une liste,
Lg une liste de termes_2,
Sint,Subst deux substitutions,
Lvt,Lvi.

pre: Lvt et Lvi sont des listes de termes (V/Term) où V est une variable_2 Vi. Si dans une liste il existe deux termes (V1/T1) et (V2/T1) alors V1 est différent de V2.

Cette procédure détermine si

Evi(Lg) est l'ensemble des variables Vi qui apparaissent dans Lg.

Lg' est la liste Lg où chaque variable Vi ∈ Evi(Lg) est remplacée par le terme t si ∃ un couple (Vi,t) ∈ Lvt.

Soit θ le mgu de (L Sint) et de (Lg' Sint), Sl=θ:var(L) et Subst=Sint.Sl

Lv' est une liste qui contient un terme (Vj/tj) pour chaque variable Vj qui apparaît dans Evi(Lg) et pas dans un terme (v/t) de Lvt. Si ∃ un lien (Vj/term) dans θ:Evi(Lg) alors Tj est tel que Tj.Subst= term, sinon Tj est une variable qui n'est ni élément de var(L.Sint) ni élément de var(Lvt).

Lvi est la concaténation de Lvt et Lv'.

****

procédure sub_to_conj(L,Ctail,C,Lv).

Soit
L une liste de termes (V/T).
Ctail,C des conjonctions de termes,
Lv une liste de variables.

Cette procédure détermine si

C=(V1=T1),...,(Vn=Tn),Ctail avec (Vi/Ti) un élément de L tel que
-Vi n'appartient pas à Lv
-il n'existe pas un terme (Vj/Tj), Vj==Vi qui précède (Vi/Ti) dans L.

Directionnalité.

In(L(ngv),any,var,L(var)) Out(nc,nc,ngv,nc). <1,1>.

****

3.5 Algorithmes et procédures logiques.

procédure: empty_Sub(S).

Procédure prolog:
```
empty_sub([]).
```

****

procédure: L_to_subs(L,Subs).

Procédure prolog:
```
L_to_subs(L,L).
```

****

procédure: member_subs(V,Term,Subs).

Procédure prolog:
```
member_subs(V,Term,Subs) :-
     member((Var/Term),Subs),
     V==T,
     !.
```

****

procédure: sub_to_conj(Sub,Ctail,C).

Procédure prolog:
```
sub_to_conj(Subs,Ctail,C) :-
     sub_to_conj(Sub,Ctail,C,[]).
```

****

procédure: sub_to_conj(L,Ctail,C,Lv).

Algorithme logique:

```
sub_to_conj(L,Ctail,C,Lv) :-
  V  L=[] & C=Ctail & Lv=[].

  V  L=[(V/Term)¦T] & member(X,Lv) &  X==V,
     sub_to_conj(T,Ctail,C,Lv).

  V  L=[(V/Term)¦T] & ¬ ∃ X : (member(X,Lv) &  X==V) &
     C=((V=Term),C1) & Lv1= [V¦Lv] &
     sub_to_conj(T,Ctail,C1,[V¦Lv]).
```

Procédure prolog:

```
sub_to_conj([(V/Term)¦T],Ctail,C,Lv) :-
                member(X,Lv),
                X==V,
                !,
                sub_to_conj(T,Ctail,C,Lv).
sub_to_conj([(V/Term)¦T],Ctail,((V=Term),C1),Lv) :-
                sub_to_conj(T,Ctail,C1,[V¦Lv]).
sub_to_conj([],Ctail,Ctail,[]).
```

****

procédure: comp(S1,S2,S).

Algorithme logique:

```
comp(S1,S2,S) <=>
          S1=[] & S=S2,
       V  S1=[H1¦T1] & H1=(V/Term) & substitue(Term,S2,Ts2)&
          S=[(V/Ts2)¦T] & comp(T1,S2,T).
```

Procédure prolog:

```
comp([],S2,S2).

comp([(V/Term)¦T1],S2,[(V/Ts2)¦T]) :-
          substitue(Term,S2,Ts2),
          comp(T1,S2,T).
```

****

procédure: substitue(T,Subst,Ts).

Algorithme logique:

```
substitue(T,Subst,Ts) <=>
          var(T) & member_subs(T,Term,Subst) & Ts=Term,

       V  var(T) &
          ¬ ∃ Term: member_subs(T,Term,Subst) & Ts=T

       V  atomic(T) & Ts=T

       V  ¬ var(T) ¬ atomic(T) & T=..[F¦L] &
          subs_L(L,Subst,Ls) & Ts=..[F¦Ls].
```

Procédure prolog:

```
substitue(T,Subst,Ts) :-
          var(T),
          member((V/Term),Subst),
          V==T,
          !,
          Ts=Term.
substitue(T,Subst,T) :-
          var(T),
          !.
substitue(T,_,T) :-
          atomic(T),
          !.
substitue(T,Subst,Ts) :-
          T=..[F|Ls],
          subs_L(L,Subst,Ls),
          Ts=..[F|Ls].
```

****

procédure: subs_L(L,Subs,Ls).

Algorithme logique:

```
subs_L(L,Subs,Ls) <=>
          L=[] & Ls=[]

       V  L=[H|T] & Ls=[Hs|Ts] &
          substitue(H,Subs,Hs) &
          subs_L(T,Subs,Ts).
```

Procédure prolog:

```
subs_L([],_,[]).

subs_L([H|T],Subs,[Hs|Ts]) :-
          substitue(H,Subs,Hs)
          subs_L(T,Subs,Ts).
```

****

procédure: unify(T1,T2,T,M).

procédure prolog:

```
unify(T1,T2,T,M) :-
     unifier(T1,T2,M),
     substitue(T1,M,T).
```

****

procédure: unifier(T1,T2,M).

Algorithme logique:

```
unifier(T1,T2,M) <=>
          var(T1) & var(T2) & T1==T2 & empty_Sub(M).

       V  var(T1) & var(T2) & M=[(T1/T2)]

       V  var(T1) & nonvar(T2) & M=[(T1/T2)]

       V  nonvar(T1) & var(T2) & M=[(T2/T1)]

       V  atomic(T1) & atomic(T2) & T1=T2 & M=[]

       V  nonvar(T1) & nonvar(T2) &
          ¬ atomic(T1) & ¬ atomic(T2) &
          T1=..[F|L1] & T2=..[F|L2] & unifier_L(L1,L2,[],M).
```

Procédure prolog:

```
unifer(T1,T2,[]) :-
          var(T1),
          var(T2),
          T1==T2,
          !.
unifier(T1,T2,[(T1/T2)]) :-
          var(T1),
          !.
unifier(T1,T2,[(T2/T1)]) :-
          var(T2),
          !.
unifier(T1,T1,[]] :-
          atomic(T1),
          !.
unifier(T1,T2,M) :-
          not(atomic(T2)).
          T1=..[F|L1],
          T2=..[F|L2],
          unifier_L(L1,L2,[],M).
```

****

procédure: unifier_L(L1,L2,M1,M).

Algorithme logique:

```
unifier_L(L1,L2,M1,M) <=>
       V  L1=[] & L2=[] & M=M1,

       V   L1=[H1|T1] & L2=[H2|T2] & unifier(H1,H2,Mh) &
          comp(M1,Mh,M2) & unifier_L(T11,T21,M2,M)
```

Procédure prolog:

```
unifier_L([],[],M,M).

unifier_L([H1|T1],[H2|T2],M1,M) :-
          unifier(H1,H2,Mh),
          comp(M1,Mh,M2),
          unifier_L(T11,T21,M2,M).
```

****

```
procédure: unifier_L2(L,Lg,Sint,Subst,Lvt,Lvi).

Algorithme logique:
unifier_L2(L,Lg,Sint,Subst,Lvt,Lvi) <=>
          L=[] & Lg=[] & Subst=Sint & Lvi=Lvt

          L=[H|T] & Lg=[Hg|Tg] &
          unifier_2(H,Hg,Sint,Sint1,Lvt,Lvt1) &
          unifier_L2(T,Tg,Sint1,Subst,Lvt1,Lvi).

************************************************************

procédure: unifier_2(T,Tg,Sint,Subst,Lvt,Lvi).

Algorithme logique:
unifier_2(T,Tg,Sint,Subst,Lvt,Lvi) <=>
          is_vi(Tg) &  member((Tg/Term),Lvt) &
          substitue(T,Sint,T1) & substitue(Term,Sint,Term1) &
          unifier(T,Term1,Mt) &
          comp(Sint,Mt,Subst) & Lvi=Lvt.

      V   is_vi(Tg) &
          ¬ ∃  Term, Sv1 : member((Tg/Term),Lvt,Sv1) &
          member((Tg/T),Lvi,Lvt) & Subst=Sint,

      V   (is_gi(Tg) V is_ni(Tg)) & Subst=Sint & Lvi=Lvint,

      V   is_const(Tg) & substitue(T,Sint,T1) &
          unifier(T1,Tg,Mt) & comp(Sint,Mt,Subst) &
          Lvt=Lvi

      V   compound(Tg) & substitue(T,Sint,T1) & var(T1) &
          functor(Tg,F,N) & functor(X,F,N) &
          unifier_2(X,Tg,[],Sx,Lvt,Lvi) &
          substitue(X,Sx,X1) & L_to_sub([T1/X1],St),
          comp(Sint,St,Subst)

       V  compound(Tg) & substitue(T,Sint,T1) & compound(T1)&
          Tg=..[F|Lg) & T=..[F|L] &
          unifier_L2(Lg,L,Sint,Subst,Lvt,Lvi).

************************************************************
```

4 Module "programme".

4.1 Spécification de l'interface.

procédure r_cl(P,Cl,P1).

Soit
P, P1 des programmes,
Cl une clause,

Cette procédure détermine si

Cl est la première clause de P et P1 est P sans cette première clause.

Directionnalité.

In(P(ngv),var,var) Out(nc,ngv,P(ngv)). <0,1>.

****

procédure w_cl(P,Cl,P1).

Soit
P, P1 deux programmes,
Cl une clause.

Cette procédure détermine si

Cl est la dernière clause de P1 et P est P1 sans cette dernière clause.

Directionnalité.

In(P(ngv),var,var) Out(Nc,ngv,P(ngv)). <1,1>.

****

procédure empty_p (Name,P).

soit
P un programme,
Name un atome.

Cette procédure détermine si

P est un programme de nom Name qui ne contient pas de clause.

effets de bord.

P est créé pour tout le programme, sa portée ne se limite pas à la clause où il a été créé.

Directionnalité.

In(any,P(ngv)) Out(gr,P(ngv)). <0,1>.
In(gr,var) Out(gr,P(ngv)). <1,1>.

4.3 Choix de représentation:

Un programme est représenté par la liste des clauses du programme. Cette liste est une liste associative.

4.4 Algorithmes et procédures logiques.

procédure: r_cl(P, Cl, P1).

Procédure prolog:

```
r_cl(P, Cl, P1) :-
                f_elem(P,Cl,P1).
```

****

procédure: w_cl(P, Cl, P1)

Procédure prolog:

```
w_cl(P,Cl,P1) :-
                add_al(P,Cl,P1).
```

****

procédure: empty_p(Name,P).

Procédure prolog:

```
empty_p(Name,P) :-
     empty_al(Name,P).
```

5. Module "descr_arbr".

5.1 Spécification de l'interface.

procédure descr_to_Lag(As,L).

Soit
- As un descr_As,
- L une liste d'atomes_2

Cette procédure détermine si

Sg est une liste qui possède un atome_2 qui décrit de manière générale l'instanciation des prédicats dont l'instanciation au moment de l'appel est décrite dans AS.

Directionnalité.

In(gr,var) Out(gr,gr). <1,1>.

****

procédure empty_descr(As).

Soit
As un descr_As.

Cette procédure détermine si

As est un decr_As vide.

Directionnalité.

In(any) Out(gr) <0,1>

****

procédure memb_inf_C(newname_C, L, As, As1).

Soit
newname_C un symbole de prédicat,
L une liste d'atomes_0,
As, As1 deux descr_As.

pré:
-Soit C la classe de noeuds similaires tels que
Newname(C)=newname_C,
-$F_{\sigma}^{*}(C)=\{s_1,\ldots,s_n\}$
-L est la liste des éléments de $F_{\sigma}^{*}(C)$ classés par ordre.
-As1 ne contient pas d'information sur la classe de noeuds similaires C.

Cette procédure détermine si

As contient l'information, (newname_C, L), à propos de l'ensemble de noeuds similaires C. As1 est As sans cette information.

Directionnalité.

In(gr,gr,var,gr) Out(gr,gr,gr,gr). <1,1>.
In(var,var,gr,var) Out(gr,gr,gr,gr). <0,1>.

****

procédure memb_inf_T(newname_C, La1, As, As1).

Soit
newname_C un symbole de prédicat,
La1 une liste d'atomes_1
As, As1 deux descr_As.

pré:
-Soit, C une classe de noeuds similaires
Newname(C)=newname_C,
-T un noeud $\in$ C, L est la liste des atomes_1 éléments de T.
-As contient de l'information sur la classe de noeuds similaires C.

Cette procédure détermine si

As contient l'information (newname_C, L) à propos du noeud T.
As1 est As sans cette information.

Directionnalité.

```
In(gr,gr,var,gr)          Out(gr,gr,gr,gr). <1,1>.
In(var,var,gr,var)  Out(gr,gr,gr,gr). <0,1>.
```

****

procédure memb_solve(S, As, As1).

Soit
S un atome_1,
As,As1 deux descr_As.

Cette procédure détermine si

As contient l'information S à propos d'un atome_0 sélectionné pour être entièrement résolu. As1 est As sans cette information.

Directionnalité.

```
In(gr,var,gr)        Out(gr,gr,gr). <1,1>.
In(var,gr,var)       Out(gr,gr,gr). <0,1>.
```

****

procédure no_inf_T(As).

Soit
As un decr_As.

Cette procédure détermine si

As ne contient d'information à propos d'aucun noeud.

Directionnalité.

```
In(gr)          Out(gr). <0,1>.
```

****

procédure no_solve(As).

Soit
As un descr_As.

Cette procédure détermine si

As ne contient de l'information à propos d'aucun atome_1 sélectionné pour être entièrement résolu.

Directionnalité.

In(gr) Out(gr). <0,1>.

****

5.2 Choix de représentation:

Un descr_As est représenté par un triple (Lt,Lc,Ls) où
-Lt est une liste de couples (Newname_C,La1),
-Lc est une liste de couples (Newname_C,LFo),
-Ls est une liste d'atomes_1 S..

5.3 Spécification des procédures internes.

procédure build_A(P,La1,La0,New_A).

Soit
P un symbole de prédicat,
La1 une liste d'atomes_1,
La0 une liste d'atomes_0,
New_A un atome_1.

pré: $\forall a_1 \in La1, \exists a_0 \in La0$ tel que $Fo(a_1)=a_0$.

Cette procédure détermine si

Soit La0=[t1,...,tn], alors New_A=P(L1,...,Ln) avec
$\forall$ i, $1 \leq i \leq n$, Li=[ti1,...,tim], la liste de tous les tij $\in$ La1, $1 \leq j \leq m$, tels que Fo(tij)=ti.

Directionnalité.

In(gr,gr,gr,var) Out(gr,gr,gr,gr). <1,1>.

****

procédure build_A_gen(P,La1,La0,New_A,Ltemp).

Soit P un symbole de prédicat,
La1 une liste d'atomes_1,
La0 une liste d'atomes_0,
New_A un atome_1.

pré: $\forall a_i \in La1, \exists a_o \in La0$ tel que $Fo(a_i)=a_o$.

Cette procédure détermine si

Soit $La0=[t_k,...,tn]$ et $Ltemp=[Lt_1,...,Lt_{k-1}]$ alors
$New\_A=P(L1,...,L_{k-1},L_k,Ln)$ avec
- $\forall$ i, $1 \leq i \leq k-1$, $L_i=Lt_i$,
- $\forall$ i, $1 \leq k \leq i \leq n$, $Li=[ti1,...,tim]$, la liste de tous les tij $\in$ La1, $1 \leq j \leq m$, tels que $Fo(tij)=ti$.

Directionnalité.

In(gr,gr,gr,var,gr) Out(gr,gr,gr,gr,gr). <1,1>.

****

procédure select_A1(L,Ao,Sel,NoSel).

Soit
L, Sel, NoSel des listes d'atomes_1,
Ao un atome_0.

Cette procédure détermine si

Sel est la liste des éléments de L tels que Fo(ti)=Ao. NoSel est la liste des éléments de L tels que Fo(ti) =/= Ao.

Directionnalité.

In(gr,gr,var,var) Out(gr,gr,gr,gr). <1,1>.

****

procédure gen_T2(T1,T2,Tg).

Soit
T1,T2,Tg des termes d'instanciation_2.

Cette procédure détermine si

Tg=Gen(T1,T2).

Directionnalité.

In(gr,gr,var) Out(gr,gr,gr). <1,1>.

****

procédure gen_list_T(L1,L2,Lg).

Soit
L1,L2,Lg des listes d'atomes_2.

Cette procédure détermine si

L1=[t_{11},...,t_{1n}], L2=[t_{21},...,t_{2n}] et Lg=[t_{g1},...,t_{gn}] avec
V i: 1≤i≤n t_{gi}=gen(t_{1i},t_{2i}).

Directionnalité.

In(gr,gr,var) Out(gr,gr,gr).<1,1>.

****

procédure des_t_l(As,Lint,L).

Soit
As un descr_As,
L,Lint des listes de couple (P,Ag),
Ag un atome_2
P le symbole de prédicat de Ag,

Cette procédure détermine si

L' est une liste qui possède un atome_2 qui décrit de manière générale l'instanciation de tous les prédicats dont l'instanciation au moment de l'appel est décrite dans AS.

La liste L est telle que :
-pour tout A,A1, A ∈ Lint et A1 ∈ L' tel que A et A1 ont même symbole de prédicat alors L contient un élement Ag=gen(A,A1).
-pour tout A ∈ Lint tel qu'il n'existe pas A1 ∈ L', A et A1 ont même symbole de prédicat alors L contient un élément A.
-pour tout A1 ∈ L' tel qu'il n'existe pas A ∈ Lint, A et A1 ont même symbole de prédicat alors L contient un élément A1.

Directionnalité.

In(gr,gr,var) Out(gr,gr,gr). <1,1>.

****

5.4 Algorithmes et procédures logiques.

procédure: descr_to_lag(As,L).

Procédure prolog:
```
descr_to_lag(As,L) :-
     des_t_l(As,[],L).
```

****

procédure: des_t_1(As,Lint,L).

Algorithme logique:

```
des_t_1(As,Lint,L) :-
      no_inf_T(As) & no_solve(As) & L=Lint.
   V  memb_inf_T(P,La1,As,As1) & memb_inf_C(P,Lo,As,As2) &
      member((P,Ag),Lint,Lt) & build_A(P,La1,Lo,New_A) &
      gen_T(Ag,New_A,Ag1) & member((P,Ag1),Lint1,Lt) &
      des_t_1(As1,Lint1,L).
   V  memb_inf_T(P,La1,As,As1) & memb_inf_C(P,Lo,As,As2) &
      ¬ ∃ Ag,Lint :member((P,Ag),Lint,Lt) &
      build_A(P,La1,Lo,New_A) &  member(Ag,Lint1,Lint) &
      des_t_1(As1,Lint1,L).
   V  memb_solve(S,As,As1) & functor(S,F,N) &
      member((F,Ag),Lint,Lt) & gen_T(S,Ag,Ag1) &
      member((F,Ag1),Lint1,Lt) &
      des_t_1(As1,Lint1,L).
   V  memb_solve(S,As,As1) & functor(S,F,N) &
      ¬ ∃ Ag,Lt :member((F,Ag),Lint,Lt) &
      member((F,S),Lint1,Lint) &
      des_t_1(As1,Lint1,L).
```

Procédure prolog:

```
des_t_1(As,Lint,L) :-
          memb_inf_T(P,La1,As,As1),
          member((P,Ag),Lint,Lt),
          !,
          memb_inf_C(P,Lo,As,_),
          build_A(P,La1,Lo,New_A),
          gen_T(Ag,New_A,Ag1),
          member((P,Ag1),Lint1,Lt),
          des_t_1(As1,Lint1,L).
des_t_1(As,Lint,L) :-
          memb_inf_T(P,La1,As,As1),
          memb_inf_C(P,Lo,As,_),
          build_A(P,La1,Lo,New_A),
          member((P,Ag),Lint1,Lint),
          des_t_1(As1,Lint1,L).
des_t_1(As,Lint,L) :-
          memb_solve(S,As,As1),
          functor(S,F,_),
          member((F,Ag),Lint,Lt),
          !,
          gen_T(S,Ag,Ag1),
          member(Ag1,Lint1,Lt),
          des_t_1(As1,Lint1,L).
des_t_1(As,Lint,L) :-
          memb_solve(S,As,As1),
          functor(S,F,_),
          member((F,S),Lint1,Lint),
          des_t_1(As1,Lint1,L).
des_t_1(([],_,[]),L,L).
```

****

procédure: empty_descr(As).

Procédure prolog:
```
empty_descr(([],[],[])).
```

****

procédure: memb_inf_C(newname_C, LFo, As, As1).

Algorithme logique:
```
memb_inf_C(newname_C, LFo, As, As1) <=>
          As=(Lt,Lc,Ls)
        & member((newname_C,LFo),Lc,Lc1),
        & As1=(Lt,Lc1,Ls).
```

Procédure prolog:
```
memb_inf_C(newname_C,L,(Lt,Lc,Ls),(Lt,Lc1,Ls)) :-
          member((newname_C,LFo),Lc,Lc1).
```

****

procédure: memb_inf_T(newname_C, La1, As, As1).

Algorithme logique:
```
memb_inf_T(newname_C, La1, As, As1) <=>
          As=(Lt,Lc,Ls)
       &  member([(newname_C,La1)],Lt,Lt1)
       &  As1=(Lt1,Lc,Ls).
```

Procédure prolog:
```
memb_inf_T(newname_C,La1,(Lt,Lc,Ls),(Lt1,Lc,Ls):-
          member((newname_C,La1),Lt,Lt1)..
```

****

procédure: memb_solve(S, As, As1)

Algorithme logique:
```
memb_solve(S, As, As1) <=>
          As=(Lt,Lc,Ls)
       &  member(S,Ls,Ls1)
       &  As1=(Lt,Lc,Ls1).
```

Procédure prolog:
```
memb_solve(S,(Lt,Lc,Ls),(Lt,Ls,Ls1) :-
          member(S,Ls,Ls1).
```

****

procédure: no_inf_T(As).

Procédure prolog:
```
no_inf_T(([],Lc,Ls).
```

****

procédure: no_solve(As).

Procédure prolog:

```
no_solve((Lt,Lc,[]).
```

****

procédure: build_A(P,La1,La0,New_A)

Algorithme prolog:

```
build_A(P,La1,La0,New_A) :-
          build_A_gen(La1,La0,Lna), New_A=..[P|Lna].
```

****

procédure: build_A_gen(La1,La0,Lna).

Algorithme logique:

```
build_A_gen(La1,La0,Lna) <=>
          La0=[] & Lna=[]

     V    La0=[ H| Ta0] & select_A1(La1,H,Sel,Nsel)
        & Lna=[Sel| Lna1] &
        & build_A_gen(Nsel,Ta0,Lna1)
```

Procédure prolog:

```
build_A_gen(_,[],[]).

build_A_gen(La1,[H|Ta0],Lna) :-
          select_A1(La1,H,Sel,Nsel),
          Lna=[Sel|Lna1],
          build_A_gen(Nsel,Ta0,Lna)
```

****

procédure: select_A1(L,Ao,Sel,NoSel).

Algorithme logique:

```
select_A1(L,Ao,Sel,NoSel) <=>
          L=[] & Sel=[] & NoSel=[]

      V   L=[H|T] & Fo(Ao,H) & Sel1=[H|Sel]
        & select_A1(T,Ao,Sel1,NoSel)

      V   L=[H|T] & ¬ Fo(Ao,H) & NoSel=[H|NoSel1]
        & select_A1(T,Ao,Sel,NoSel1).
```

Procédure prolog:

```
select_A1([],_,[],[]).
select_A1([H|T],Ao,[H|Sel1],NoSel) :-
          Fo(Ao,H),
          !,
          select_A1(T,Ao,Sel1,NoSel).
select_A1([H|T],Ao,Sel,[H|NoSel1]) :-
          select_A1(T,Ao,Sel,NoSel1).
```

****

procédure: gen(T1,T2,Tg).

Algorithme logique:

```
gen_T(T1,T2,Tg) <=>
          is_vi(T1) & is_vi(T2) & f_vi(T1,T2,Tg)

        V  is_vi(T1) & ¬ is_vi(T2) & is_ni(Tg),

        V  ¬ is_vi(T1) & is_vi(T2) & is_ni(Tg),

        V   is_gi(T1)  & fo(T2,g) & is_gi(Tg)

        V   fo(T1,g) & is_gi(T2) & is_gi(Tg)

        V   is_cont(T1) & is_cont(T2) & T1=T2 & Tg=T1

        V  is_gi(T1)  & ¬ fo(T2,g) & is_ni(Tg)

        V  ¬ fo(T1,g)& is_gi(T2)  & is_ni(Tg)

        V  is_ni(T1)  & is_ni(Tg)

        V  is_ni(T2)  & is_ni(Tg)

        V  compound(T1) & compound(T2) &
           T1=..[F|L1] & T2=..[F|L2] & Tg=..[F|Lg] &
           gen_list_T(L1,L2,Lg).
```

Procédure prolog:

```
gen_T(T1,T2,Tg) :-
                is_vi(T1),
                is_vi(T2),
                f_vi(T1,T2,Tg),
                !.
gen_T(any,_,any) :-
                !.
gen_T(_,any,any) :-
                !.
gen_T(g,T2,g) :-
                fo(T2,g),
                !.
gen_T(T1,g,g) :-
                fo(T1,g),
                !.
gen_T(T1,T1,T1) :-
                atomic(T1).
gen_T(T1,T2,Tg) :-
                functor(T1,F,N),
                functor(T2,F,N),
                !.
                T1=..[F|L1],
                T2=..[F|L2],
                gen_list_T(L1,L2,Lg).
                Tg=..[F|Lg]
gen_T(_,_,any).
```

****

```
procédure: gen_list_T(L1,L2,Lg).

Algorithme logique:
gen_list_T(L1,L2,Lg) <=>
          L1=[] & L2=[] & Lg=[]

       V  L1=[H1|T1] & L2=[H2|T2] & Lg=[Hg|Tg] &
          gen_T(H1,H2,Hg) & gen_list_T(T1,T2,Tg).

Procédure prolog:
gen_list_T([],[],[]).
gen_list_T([H1|T1],[H2|T2],[Hg|Tg]) :-
          gen_T(H1,H2,Hg),
          gen_list_T(T1,T2,Tg).

**************************************************************
```

6. Module "fonction de base".

6.1 Spécification de l'interface.

```
procédure  f_ndv(P,Subst,P1,Fp,Mode).

Soit
     P,Fp des prédicats,
     P1 un atome_2
     Subst une substitution,
     Mode une déclaration de mode.

Cette procédure détermine si

Fp=ndv(flat((P Subst),P1), flat(P1)).

Mode = de_mode(T1).

Directionnalité.

In(ngv,S(ngv),gr,var,var)   Out(ngv,S(ngv),gr,ngv,gr). <0,1>.
```

6.2 Spécification des procédures internes.

```
procédure  f_ndv_L(L,Subst,L1,Fl,Lmode,Lvint,Lvi).

Soit
     L,Fl,Lmode des listes,
     L1 une liste de termes_2,
     Subst une substitution.
```

Cette procédure détermine si

```
Fl=ndv_Lt(flat_Lt((L Subst),L1), flat_Lt(L1), Lvint).
```

Lmode est la liste ndv_Lt(flat_Lt(L1),Lvi) où toute occurrence d'une variable Vj est remplacée par o, Gj ou une constante par i, Nj par ?.

Lvi est la liste des variables_2 Vi qui apparaissent dans ndv_Lt(flat_Lt(L1),Lvi) concaténée à Lvint.

Directionnalité.

```
In(l(ngv),S(ngv),gr,var,var,gr,var)
Out(L(ngv),S(ngv),gr,L(ngv),gr,gr,gr). <0,1>.
```

****

6.3 Algorithmes et procédures logiques.

procédure: fl_ndv(P,Subst,P1,Fp,Mode).

Algorithme logique:

```
fl_ndv(P,Subst,P1,Fp,Mode) <=>
     P=..[F¦L] & P1=..[F¦L1] &
     f_ndv_L(L,Subst,L1,Fl,Lmode,[],Lvi) &
     Fp=..[F¦Fl] & Mode=..[F¦Lmode].
```

****

procédure: f_ndv_L(L,Subst,L1,Fl,Lmode,Lvint,Lvi).

Algorithme logique:

```
f_ndv_L(L,Subst,L1,Fl,Lmode,Lvint,Lvi) <=>
     L=[] & L1=[] & FL=[] & Lmode=[] & Lvi=Lvint.

  V  L=[H0¦T] & L1=[H1¦T1] &
     substitue(H0,Subst,H) &
     compound(H) & compound(H1) &
     H=..[F¦Lh] & H1=..[F¦Lh1] & empty_sub(S) &
     f_ndv_L(Lh,S,Lh1,Fh,Hmode,Lvint,Lvint1) &
     f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint1,Lvi) &
     append(Fh,Ft,Fl) & append(Hmode,Tmode,Lmode).

  V  L=[H¦T] & L1=[H1¦T1] &
     is_vi(H1) & member(H1,Lvint) &
     f_ndv_L(T,Subst,T1,Fl,Fmode,Lvint,Lvi)

  V  L=[H0¦T] & L1=[H1¦T1] &
     substitue(H0,Subst,H) &
     is_vi(H1) & ¬ member(H1,Lvint) &
     Fl=[H¦Ft] & Lmode=[o¦Tmode] & Lvint1=[H1¦Lvint] &
     f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint1,Lvi)

  V  L=[H0¦T] & L1=[H1¦T1] &
     substitue(H0,Subst,H) &
     (is_Gi(H1) V is_const(H1)) &
     Fl=[H¦Ft] & Lmode=['I'¦Tmode] &
     f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi)
```

```
  V  L=[H0|T] & L1=[H1|T1] &
     substitue(H0,Subst,H) &
     is_Ni(H1)
     Fl=[H|Ft] & Lmode=[?|Tmode] &
     f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi)
```

Procédure prolog:

```
f_ndv_L([H|T],Subst,[H1|T1],Fl,Lmode,Lvint,Lvi) :-
          is_vi(H1),
          member(H1,Lvint),
          !,
          f_ndv_L(T,Subst,T1,Fl,Fmode,Lvint,Lvi).
f_ndv_L([H0|T],Subst,[H1|T1],Fl,Lmode,Lvint,Lvi) :-
          is_vi(H1),
          !,
          substitue(H0,Subst,H),
          Fl=[H|Ft],
          Lmode=[o|Tmode],
          Lvint1=[H1|Lvint],
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint1,Lvi).
f_ndv_L([H0|T],Subst,[any|T1],Fl,Lmode,Lvint,Lvi) :-
          !,
          substitue(H0,Subst,H),
          Fl=[H|Ft],
          Lmode=[?|Tmode],
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi).
f_ndv_L([H0|T],Subst,[H1|T1],Fl,Lmode,Lvint,Lvi) :-
          atomic(H1),
          !,
          substitue(H0,Subst,H),
          Fl=[H|Ft],
          Lmode=[i|Tmode],
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint,Lvi).
f_ndv_L([H0|T],Subst,[H1|T1],Fl,Lmode,Lvint,Lvi) :-
          substitue(H0,Subst,H),
          H=..[F|Lh] & H1=..[F|Lh1],
          f_ndv_L(Lh,[],Lh1,Fh,Hmode,Lvint,Lvint1),
          append(Fh,Ft,Fl),
          append(Hmode,Tmode,Lmode),
          f_ndv_L(T,Subst,T1,Ft,Tmode,Lvint1,Lvi).
f_ndv_L([],_,[],[],[],Lvi,Lvi).
```

****

7. Module "transformation programme".

7.1 Spécification de l'interface.

procédure gen_mode(P,As,Pnew,Name,Mode).

Soit
P,Pnew des programmes,
As un descr_arb,
Name un atome,
Mode une liste de déclarations de mode.

Cette procédure détermine si

Sg est une liste qui possède un atome_2 qui décrit de manière générale l'instanciation des prédicats dont l'instanciation au moment de l'appel est décrite dans AS.

Pnew est un programme de nom Name dont les clauses sont les clauses de P transformées en fonction de Sg. Mode est la liste des déclarations de mode pour Pnew.

Directionnalité.

In(P(ngv),gr,var,var) Out(P(ngv),gr,P(ngv),gr). <1,1>.

7.2 Spécification des procédures internes.

procédure trans_p(P,Sg,Pnew,Name,Mode).

Soit
P,Pnew des programmes,
Sg une liste d'atomes_2
Name un atome,
Mode une liste.

pre:
-Sg ne contient pas deux atomes_2 Ag et Ag1 tels que Ag et Ag1 ont même symbole de prédicat.

Cette procédure détermine si

Pnew est un programme de nom P dont les clauses sont les clauses de P transformées en fonction de Sg. Si Sg=[A1,...,An] alors Mode=[M1,...,Mn] avec Mi=de_mode(Ai), $1 \leq i \leq n$.

Directionnalité.

In(P(ngv),gr,var,var) Out(P(ngv),gr,P(ngv),gr). <1,1>.

****

procédure tr_p(P,Sg,Pint,Pnew,Mint,Mode).

Soit
P,Pint,Pnew des programmes,
Sg une liste d'atomes_2
Mint,Mode deux listes.

pre:
-Sg ne contient pas deux atomes_2 Ag et Ag1 tels que Ag et Ag1 ont même symbole de prédicat.

Cette procédure détermine si

P' est P où chaque clause est transformée en fonction de Sg.

Pnew est le programme Pint auquel on a ajouté les clauses de P'.

si Sg contient un atome_2 Ag et que dans le programme P il existe un prédicat avec même foncteur et même arité, alors Mode' contient un prédicat M=de_mode(Ag).

Mode est la liste Mint a laquelle ont été ajoutés les éléments de Mode' qui n'y figurent pas encore.

Mode est la liste des éléments de l'union des éléments de Mint et de Mode'

Directionnalité.

In(P(ngv),gr,P(ngv),var,gr,var) Out(nc),gr,nc,P(ngv)gr,gr,).
<1,1>.

****

procédure trans_cl(CL,Sg,Clnew,Mode).

Soit
Cl,Cl_new des clauses,
Sg un ensemble d'atomes_0,
Mode une liste.

Cette procédure détermine si

Clnew=trans_cl(Cl).

Si ∃ Ag ∈ Sg tel que Ag a même symbole de prédicat que la tête de Cl alors Mode=[de_mode(Ag)], sinon Mode=[].

Directionnalité.

In(ngv,gr,var,var) Out(nc,gr,ngv,gr). <1,1>.

****

procédure trans_tail(T,Sg,New_tail,Subst).

Soit
T,New_tail des conjonctions de prédicats,
Sg un ensemble d'atomes_2,
Subst une substitution.

Cette procédure détermine si

New_tail=trans_conj(T Subst,Sg).

Directionnalité.

In(ngv,gr,var,S(ngv)) Out(ngv,gr,ngv,S(ngv)). <1,1>.

****

procédure trans_pred(P,Sg,Pnew,S1,Subst,Lmode).

Soit
P et Pnew des prédicats,
Sg un ensemble d'atomes_2,
Subst une substitution
Lmode une liste.

Cette procédure détermine si

Pnew=tr_pred(P,Sg).

S'il existe Ag ∈ Sg tel que Ag a même symbole de prédicat que P alors Lmode=[de_mode(Ag)], sinon Lmode=[], si Ag s'unifie à P avec mgu θ alors Subst= S1.θ:var(P)

Directionnalité.

In(ngv,gr,var,S(ngv),var,var) Out(ngv,gr,ngv,S(ngv),S(ngv),gr). <1,1>.

****

7.3 Algorithmes et procédures logiques.

procédure: gen_mode(P,As,Pnew,Name,Mode).

Procédure prolog:

```
gen_mode(P,As,Pnew,Name,Mode) :-
     descr_to_Lag(As,Sg),
     trans_p(P,Sg,Pnew,Name,Mode).
```

****

procédure: trans_p(P,Sg,Pnew,Mode).

Algorithme logique:

```
trans_p(P,Sg,Pnew,NameMode) <=>
      empty_p(Name,Pint) &
      tr_p(P,Sg,Pint,Pnew,[],Mode).
```

Procédure prolog:

```
trans_p(P,Sg,Pnew,Name,Mode) :-
      empty_p(Name,Pint),
      tr_p(P,Sg,Pint,Pnew,[],Mode).
```

****

procédure: tr_p(P,Sg,Pint,Pnew,Mint,Mode).

Algorithme logique:

```
tr_p(P,Sg,Pint,Pnew,Mint,Mode) <=>
      empty_P(Nx,P) & Pnew=Pint et Mode=Mint.

  V    r_cl(P,Cl,P1) & trans_cl(Cl,Sg,Clnew,Mcl) &
      add_mode(Mint,Mcl,Mint1) & w_cl(Pint,Clnew,Pint1),
       tr_p(P1,Sg,Pint1,Pnew,Mint1,Mode).
```

Procédure prolog:

```
tr_p(P,Sg,Pint,Pnew,Mint,Mode):-
       r_cl(P,Cl,P1),
       trans_cl(Cl,Sg,Clnew,Mcl),
      !,
      add_S(Mint,Mcl,Mint1),
       w_cl(Pint,Clnew,Pint1),
       tr_p(P1,Sg,Pint1,Pnew,Mint1,Mode).
tr_p(P,_,Pnew,Pnew,Mode,Mode) :-
              empty_P(_,P).
```

****

procédure: trans_cl(CL,Sg,Clnew,Mode).

Algorithme logique:

```
trans_cl(CL,Sg,Clnew,Mode) <=>
       Cl=(H:-T) & trans_pred(H,Sg,Htrans,[],Subst,Mode) &
     & Clnew=(Htrans:-Ttrans) & trans_tail(T,Sg,Ttrans,Subst)

  V  ¬ ∃ H,T Cl=(H:-T) &
       trans_pred(H,Sg,Htrans,[],S,Mode).
```

Procédure prolog:

```
trans_cl((H :- T),Sg,(Htrans:-Ttrans),Mode) :-
       !,
       trans_pred(H,Sg,Htrans,[],Subst,Mode),
       trans_tail( T,Sg,Ttrans,Subst).
trans_cl(H,Sg,Htrans,Mode) :-
       trans_pred(H,Sg,Htrans,[],_,Mode.
```

****

procédure: trans_tail(T,Sg,New_tail,Subst).

Algorithme logique:

```
trans_tail(T,Sg,New_tail,Subst) :-
      T=(A,B) &  substitue(A,Subst,As) &
      trans_pred(As,Sg,As1,[],Sa,Mode)  &
      sub_to_conj(Sa,(As1,B1),New_tail) &
      comp(Subst,Sa,Subst1)  & trans_tail(B,Sg,B1,Subst1)

  V  ¬ ∃ A1,B1: T=(A1,B1) & T=A &
     substitue(A,Subst,As) & trans_pred(As,Sg,As1,[],Sa,Mode),
      sub_to_conj(Sa,As1,New_tail).
```

Procédure prolog:

```
trans_tail((A,B),Sg,New_tail,Subst) :-
      !,
      substitue(A,Subst,As),
      trans_pred(As,Sg,As1,[],Sa,_),
      sub_to_conj(Sa,(As1,B1),New_tail,[]),
      comp(Subst,Sa,Subst1),
      trans_tail(B,Sg,B1,Subst1).
trans_tail(A,Sg,New_tail,Subst) :-
      substitue(A,Subst,As),
      trans_pred(As, Sg, As1,[],Sa,_),
      sub_to_conj(Sa,As1,New_tail,[]).
```

****

procédure: trans_pred(P,Sg,Pnew,S1,Subst,Lmode).

Algorithme logique:

```
trans_pred(P,Sg,Pnew,S1,Subst,Lmode) <=>
      functor(P,F,N) & member((F,Ag),Sg) &
      unifier_2(P,Ag,S1,Subst) & Lmode=Mode &
      fl_ndv(P,Subst,Ag,Pnew,Mode).

  V  functor(P,F,N) & ¬ ∃ Ag: member((F,Ag),Sg) &
     Lmode=[] & Subst=S1 & Pnew=P.
```

Procédure prolog:

```
trans_pred(P,Sg,Pnew,S1,Subst,Mode) :-
      functor(P,F,N),
      member((F,Ag),Sg),
      !,
      unifier_2(P,Ag,S1,Subst),
      fl_ndv(P,Subst,Ag,Pnew,Mode).
trans_pred(P,_,P,Subst,Subst,[]).
```

****

Le programme réalisé, gen_mode(P,As,Pnew,Name,Mode), permet de générer des déclarations de mode pour un programme prolog P obtenu par compilation d'informations de contrôle, à partir d'un programme dont l'arbre de résolution symbolique est décrit par **As**. **Pnew** est le programme P transformé pour pouvoir faire les déclarations de mode **Mode** et porte le nom **Name**.

La spécification complète de **gen_mode(P,As,Pnew,Name,Mode)** se trouve dans le module "transformation programme".

P et Pnew sont représentés par types abstraits. Ils peuvent être manipulés à l'aide des procédures d'interface, **empty_p/2**, **r_cl/3**, **w_cl/3**, qui permettent de tester si un programme est vide, de lire une clause et d'écrire une clause d'un programme. Ces procédures sont spécifiées plus en détails dans l'interface du module "programmes".

L'arbre de résolution symbolique est décrit par un descripteur d'arbres de résolution symbolique (descr_As) As. Un descr_As est représenté par types abstraits. Ce descr_As doit contenir une description des classes de noeuds similaires de l'arbre, les noeuds de l'arbre, et les atomes_1 sélectionnés pour être entièrement résolus.

Les procédures suivantes, spécifiées plus précisément dans l'interface du module "descr_As" permettent de créer un descr_As :

- **empty_descr(As)** permet de créer un descr_As vide.
- **memb_inf_C(newname_C, L, As, As1)** permet d'ajouter à As l'information (newname_C, L) à propos d'une classe de noeuds similaires. Newname_C est le symbole de prédicat associé à la classe de noeuds similaires et L est la liste des éléments de Fo*(C) classés par ordre.
- **memb_inf_T(newname_c,La1,As,As1)** permet d'ajouter à As l'information (newname_c,L) à propos d'un noeud de l'arbre. Newname_C est le symbole de prédicat associé à la classe de noeuds similaires à laquelle le noeud appartient et La1 et la liste des atomes_1 qui font parties du noeud.
- **memb_solve(S,As,As1)** permet d'ajouter à As, l'atome_1 S sélectionné pour être entièrement résolu..

Mode est une liste de déclarations de mode.